Corba ryhmäohjelmien toteuttamisvälineenä

Jyri Jokinen

Tampereen yliopisto
Tietojenkäsittelyopin laitos
Laudaturtutkielma
3.12.1999

Kiitokset

Aloitin tämän tutkielmani kirjoittamisen keväällä 1997, jolloin silloisessa työpaikassani TeamWARE Groupissa tutkittiin uusia tapoja tehdä asioita. Aihetta ehdotti silloinen esimieheni Antero Aalto ja työtä ohjaamaan valikoitui Kai Koskimies.

Alkuun kirjoittaminen oli helppoa, mutta sitten iski kirjoittajan blokki, joka paheni entisestään, kun työni ohjaaja jäi virkavapaalle. Lähes vuoden verran työni vain odotti uutta inspiraation aaltoa. Tämä aalto iski jälleen vuoden 1998 syksyllä, jolloin keksin valita agenttiohjelmoinnin kurssin harjoitustyöhön Corbaa sivuavan aiheen. Tutkielmaa tarjoutui ohjaamaan Roope Raisamo, jonka valvovan silmän alla sain työni kunnialliseen päätökseen.

Olen suuressa kiitollisuudenvelassa joukolle ihmisiä, jotka ovat avustaneet minua työllään, kommenteillaan tai henkisesti tukemalla:

Antero Aaltoa haluan kiittää alkuperäisestä työn ideasta sekä hienoisesta painostuksesta, jonka avulla työ eteni hiukan silloinkin, kun ideat tuntuivat muuten täysin loppuneen.

TeamWARE Groupin työntekijöistä kiitoksen ansaitsevat Jyrki Aarnos ja Juhani Palmu, joilta sain hyviä kommentteja ja ideoita työn sivujuonteiksi. Ilman Harri Toijalalta saamaani rajapintatiedostoa en olisi varmaan koskaan saanut kokeilusovellustani valmiiksi, siitä kiitos.

Yliopistolla sain apua Saila Ovaskalta sekä tietysti Roope Raisamolta, kiitos myös teille.

Lopuksi haluan kiittää kihlattuani Elina Selkälää, joka käytti ilmeisen onnistuneesti kaikkia keinoja varmistaakseen, että tämä työ ei päätynyt keskeneräisten tarujen kirjaan.

Tampereella 7.5.1999

Tampereen yliopisto Tietojenkäsittelyopin laitos
Jokinen, Jyri Nestori: Corba ryhmäohjelmien toteuttamisvälineenä
Laudaturtutkielma, 58 sivua
Toukokuu 1999

Tiivistelmä

Tämän tutkielman tarkoituksena on esitellä Corba-niminen välittäjäohjelmisto ja tutkia sen käyttökelpoisuutta ryhmäohjelmien toteuttamisvälineenä.

Corbasta on kirjoitettu suuri joukko kirjoja ja sen käyttöä on perusteltu lähinnä vertaamalla sitä muihin vastaaviin järjestelmiin, mutta sen toimivuutta nimenomaan ryhmäohjelmien toteuttamiseen ei ole tutkittu tarkemmin.

Tutkimus on suoritettu kirjallisuustutkimuksena minkä lisäksi tutkin Corban toimivuutta käytännössä toteuttamalla yksinkertaisen asiakas-palvelin-sovelluksen.

Totesin, että Corba on aidosti oliopohjaisen järjestelmän määritelmä, mikä auttaa oliopohjaisten sovellusten laatimisessa; Corban tapa erottaa palveluiden rajapinta niiden toteutuksesta pakottaa huolellisuuteen suunnitteluvaiheessa. Havaitsin myös, että Corban avulla tietoliikenteen toteuttaminen sovelluksiin on mahdollista hyvin vähällä vaivalla. Corba määrittelee myös lukuisan joukon palveluita, joiden avulla voidaan helpottaa sovelluskehittäjän työtä.

Näistä piirteistä johtuen Corba sopii hyvin ryhmäohjelmien toteuttamisvälineeksi ja sen käyttö on kannattavaa yleisemminkin hajautettujen järjestelmien pohjana.

Sisällys :

1. Johdanto

Verkottuminen on muuttamassa tietotekniikan maailmaa. Aikaisemmin toimistojen pöytäkoneet ja pankkien ja vakuutusyhtiöiden suurkoneet eivät välttämättä tienneet toisistaan mitään, mutta internetin myötä tilanne muuttuu.

Muutoksen hidasteena on kuitenkin heterogeenisyys: laitealustoja, käyttöjärjestelmiä, sovelluksia ja ohjelmointikieliä on niin paljon ja niin erilaisia, että niiden välinen kommunikointi ilman yhteistä säveltä on mahdotonta. Ongelman ratkaisemiseksi on kehitetty erilaisia hajautettujen järjestelmien arkkitehtuureja, jotka määrittelevät tällaisen sävelen.

Yksikään ratkaisu ei ole kuitenkaan saanut taakseen niin yhtenäistä laite- ja ohjelmistovalmistajien rintamaa kuin Corba. Corba – *Common Object Request Broker Architecture* – on määritelmä järjestelmästä, jossa erilliset oliot voivat sijaita toisistaan riippumatta verkkoympäristössä ja kommunikoida toistensa kanssa. Tämän standardin mukaisessa järjestelmässä on kääntäjä, joka osaa kääntää oliomäärittelykielellä tehdyn oliokuvauksen varsinaiselle ohjelmointikielelle, sekä välittäjäohjelmisto, joka osaa ottaa vastaan metodikutsut tai niiden paluuarvot ja siirtää ne sovellukselta toiselle; eri kielellä tehtyjen ohjelmien, eri tietokoneiden, vaikka eri maiden välillä.

Pelkän sovellusalustan lisäksi Corba-määritelmään liittyy myös lukuisa joukko erilaisten hyödykeolioiden määritelmiä. Näitä hyödykeolioita on määritelty kaupan, pankkitoiminnan, terveydenhuollon ja monelta muulta alalta. Ryhmäohjelmien alueelta määritelmiä on tehty jo mm. sähköposti- ja *workflow*-palveluiden toteuttamiseen. Tämän tutkielman keskeisenä aiheena on ryhmäkalenteri.

Tutkielmassani pyrin tarjoamaan lukijalle ytimekkään yleiskuvan Corbasta sekä kuvailemaan hiukan sen tärkeimpiä osia. Loppupuoliskolla annan konkreettisen kuvauksen siitä, mitä Corban käyttäminen nykyisillä välineillä käytännössä on.

Työni koostuu viidestä luvusta sekä viiteluettelosta ja sanastosta.

Toisessa luvussa valotan hiukan Corban historiaa ja taustoja sekä kerron myös miksi Corbaa kannattaa alkaa käyttää. Esitän lisäksi muutamia syitä, minkä takia Corba ja Java ovat verraton yhdistelmä. Lopuksi esittelen joitakin Corban varteenotettavia kilpailijoita sekä selitän, mitä etuja tai puutteita niissä on Corbaan verrattuna.

Kolmannessa luvussa paneudun syvemmin Corban rakenteeseen. Selitän siinä Corban eri osat ja niiden merkityksen. Corba-määritelmä on itsessään lähes tuhatsivuinen opus minkä lisäksi Corbaan olennaisesti liittyvät yleiset palvelut muodostavat toisen tuhatsivuisen opuksen. Näin ollen en ymmärrettävästikään voi käytetyssä tilassa kuvailla kaikkea perusteellisesti. Kyseessä onkin vain lyhykäinen johdanto, jonka pohjalta on mahdollista lähteä tutustumaan aiheeseen tarkemmin.

Neljännessä luvussa esittelen aiheeseen liittyvää työtäni, kalenteripalvelun oliototeutusta Corbaa käyttäen. Pohjana työlleni on *TeamWARE Calendar v. 5.2* ja siihen liittyvä C-rajapinta sekä oma oliomallini. Luvussa analysoidaan myös joitakin Corba-kehityksessä käytettäviä työkaluja.

Viides luku on yhteenveto tutkielmassani esittämistäni asioista. Tähän lukuun olen koonnut tekemäni havainnot ja yritän niiden pohjalta vastata tutkimusongelmaani: onko Corban käyttö mielekästä ryhmäohjelmien toteuttamisessa.

Liitteenä on sanasto, jossa selitetään tässä työssä käytetyt hajautettuihin järjestelmiin liittyvät termit.

Toivon, että tästä tutkielmasta on apua Corbasta kiinnostuneille tai sen käyttöönottoa harkitseville tietotekniikka-alan harrastajille.

2. Corban perusteita

2.1. Miksi Corba?

Ohjelmistotuotannon erääksi vakavaksi ongelmaksi on jo kauan tiedetty ohjelmistojen yhteensopimattomuus. Laitteistot, käyttöjärjestelmät ja ohjelmointikielet kehittyvät huimaavalla vauhdilla. Tästä syystä ohjelmistojen vanhentuminen usein johtaa niiden kehittämiseen ja ylläpitoon käytettyjen resurssien hukkaan heittämiseen, sillä vanhojen ja uusien komponenttien yhdistäminen ei useinkaan ole kannattavaa.

Yhtenä pelastajana tähän ohjelmistojen kriisiin on yhä enenevässä määrin tarjottu oliomenetelmien käyttöä.

OMG eli Object Management Group perustettiin vuonna 1989. Sen tavoitteena on alusta asti ollut edistää oliotekniikoiden käyttöä ohjelmistotyössä ja näiden avulla edistää ohjelmistojen uudelleenkäytettävyyttä, siirrettävyyttä sekä olio-ohjelmien välistä yhteistyötä hajautetuissa heterogeenisissä ympäristöissä. Näihin tavoitteisiin pyritään luomalla määritelmiä, joihin tukeutumalla ohjelmistot saadaan toimimaan keskenään laitteistoista ja käyttöjärjestelmistä riippumatta. Organisaatioon kuuluu yli 500 jäsentä: mukana on sekä ohjelmistojen tuottajia että käyttäjiä. [CORBA, 96, s.1]

Hajautettujen järjestelmien merkitys on kasvanut kun tietoverkot ovat tulleet yhä yleisemmiksi ja Internetin suosio on kasvanut. Näiden järjestelmien kehittämiseen on luotu useita systeemejä, kuten *Open Software Foundationin* DCE (*Distributed Computing Environment*), Javaa varten kehitetty RMI (*Remote Method Invocation*), Microsoftin DCOM (*Distributed Component Object Model*) sekä Corba.

Syksyllä 1990 OMG julkaisi *Object Management Architecture Guiden*. OMA tarjoaa käsitteellisen infrastruktuurin, johon kaikki OMG:n määritelmät perustuvat [CORBA, 96, s.1]. Vuonna 1991 ilmestyneessä Corba 1.1 -spesifikaatiossa määriteltiin IDL-kieli ja sen suhde eri ohjelmointikieliin sekä ORB-ohjelmointirajapinnat. Joulukuussa 1994 julkaistiin Corba 2.0 -

spesifikaatio, jossa määriteltiin eri ORB-toteutusten välinen kommunikointi. [Orfali & Harkey, 98, s. 7]

Corbaa voidaan käyttää apuna ratkaistaessa aiemmin mainittuja ongelmia. Corban avulla voidaan tietokoneet ja sovellukset yhdistää toisiinsa puuttumatta olemassa olevaan laitteisto-, verkko- ja ohjelmistoinfrastruktuuriin [Otte *et al*., 96, s. 1–3].

Corba ORB on välittäjäohjelma, eli se välittää metodikutsuja, parametreja ja binääridataa asiakassovellusten ja palvelinolioiden välillä. ORB on myös verrattain hyvä välittäjäohjelma, koska kaikki Corba ORB:t tarjoavat ainakin seuraavat edut [Orfali *et al*., 97, s. 8–9]:

- Staattiset ja dynaamiset metodikutsut
- Sidonnat korkeamman tason ohjelmointikieliin
- Itsekuvaava järjestelmä
- Ero paikallisten ja etäisten olioiden välillä on näkymätön
- Monimuotoisuus
- Yhteensopivuus olemassa olevien järjestelmien kanssa

Mitä näillä ominaisuuksilla sitten tarkoitetaan?

Staattiset ja dynaamiset metodikutsut tarkoittavat sitä, että Corba-välittäjät mahdollistavat olioiden rajapinnan kiinteän määrittelemisen, jolloin käytössä on vahva tyyppitarkastus. Toisaalta oliot voivat määritellä rajapintansa dynaamisesti myös ajon aikana ja asiakassovellukset voivat selvittää nämä rajapinnat, millä saavutetaan mahdollisimman suuri joustavuus.

Dynaamiset metodit eivät tarkoita samaa kuin olio-ohjelmoinnista tuttu dynaaminen sidonta, jossa metodit voidaan kyllä liittää niiden kutsuihin vasta ajon aikana, mutta jossa kutsuvan luokan täytyy kuitenkin tuntea kutsuttavien luokkien rajapinnat.

Sidonnat korkeamman tason ohjelmointikieliin merkitsee, että Corba-välittäjät antavat ohjelmoijan käyttää haluamaansa korkeamman tason ohjelmointikieltä. Koska olioiden rajapinnat on määritelty ohjelmointikielestä riippumattomasti, voidaan toteutukset kirjoittaa eri kielellä kuin rajapintaa

käyttävät sovellukset. Kun toteutus on näin aidosti erotettu rajapinnasta, voidaan toteutukseen tehdä muutoksia ja kutsuvaan koodiin ei tarvitse koskea lainkaan.

Järjestelmä on itsekuvaava, koska jokaisen Corba-välittäjän täytyy toteuttaa ns. rajapintatietokanta, joka sisältää ajonaikaisen tiedon jokaisen välitetyn olion rajapinnasta. Asiakasohjelmat käyttävät välittäjän tarjoamaa metatietoa selvittääkseen, kuinka palvelinolioita kutsutaan. Tämä metatieto tuotetaan automaattisesti IDL-koodista. On myös olemassa kääntäjiä, jotka kääntävät olemassa olevan ohjelmakoodin IDL:ksi esimerkiksi C++:sta tai Javasta.

Eroa paikallisten ja etäolioiden välillä voidaan sanoa näkymättömäksi, koska oliovälitin voi toimia yksinään esimerkiksi sylimikrossa, mutta yhtä hyvin se voidaan kytkeä mihin tahansa verkkoon liitettyyn tietokoneeseen esimerkiksi Internetin kautta käyttäen Corba 2.0 -standardin *Internet Inter-ORB Protocol* (IIOP) -palvelua. Välitin voi välittää olioita yksittäisen prosessin sisällä, samassa koneessa eri prosessien välillä tai useiden eri koneiden ja käyttöjärjestelmien välillä tietoverkoissa. Kutsuvan ohjelman kannalta näillä ei ole mitään eroa. Corba pitää siis huolen muuttujien tyypeistä, verkkoyhteyksistä, palvelinolioiden sijainneista ja muista vastaavista seikoista.

Corba-järjestelmää on **monimuotoinen**, koska toisin kuin RPC-pohjaiset järjestelmät (*Remote Procedure Call*), Corba-välittäjät eivät ainoastaan kutsu etäfunktioita, vaan ne kutsuvat tietyn olion metodia. Näin sama funktio voi tuottaa erilaisia tuloksia riippuen siitä, minkä olion funktiota kutsutaan.

Yhteensopivuus olemassa olevien järjestelmien kanssa johtuu siitä, että Corba erottaa rajapinnan määrittelyn ja toteutuksen niin täydellisesti toisistaan, että jo olemassa olevien sovellusten päälle on helppo luoda rajapinta muiden sovellusten kutsuttavaksi. Corban IDL-kielellä määritellään oliorajapinta, jonka toteutus tehdään esimerkiksi vanhan COBOLilla ohjelmoidun sovelluksen päälle. Näin Corba mahdollistaa kehittyvät ratkaisut. Uudet sovellukset voidaan kirjoittaa aitoina olioina ja olemassa olevista sovelluksista

tehdään oliomalli, joka määritellään käyttäen IDL-määrittelykieltä ja toteutetaan sitten alkuperäisellä sovelluksen ohjelmointikielellä.

2.2. Corba ja Java

Robert Orfalilla, Dan Harkeylla ja Jeri Edwardsilla on kaikissa kirjoissaan yksi teema, joka liittyy Corban ja Javan yhdistymiseen 'olioverkossa'. Tämä olioverkko on kirjoittajien mielestä se tekijä, joka mahdollistaa Corban lopullisen läpimurron [Orfali *et al*., 97, s. 29].

Olioverkko tarkoittaa hajautettujen olioiden ja WWW:n yhdistymistä, jota ovat puuhaamassa sellaiset tietotekniikka-alan mahtiyritykset kuin Sun, JavaSoft, IBM, Netscape, Apple, Oracle ja Hewlett Packard. Nämä ovat valinneet Corba IIOP:n hajautettujen olioiden yhteydenpitovälineeksi Internetissä ja intraneteissä. Näin Corbasta saattaa ajan mittaan tulla yhtä yleiskäyttöinen kuin TCP/IP:stä.

Mutta miksi Corba ja Java ovat niin lyömätön yhdistelmä? Tätä kysymystä kannattaa tutkailla hieman.

2.2.1. Miten Corba hyödyttää Javaa

Java yksinäänkin on erinomainen ohjelmointikieli ja ympäristö. Sen tärkein ominaisuus on laitteistoriippumattomuus, jonka ansiosta kerran käännetty ohjelma voidaan periaatteessa suorittaa missä tahansa ympäristössä, johon on toteutettu Javan virtuaalikone.

Myöskään hajautettu tietojenkäsittely ei tuota ongelmia, sillä JDK 1.1:stä alkaen Javasoft on liittänyt Javaan kiinteänä osana oman oliovälittäjänsä, RMI:n (Remote Method Invocation). RMI mahdollistaa olioiden hajauttamisen verkkoon ja niiden kutsumisen lähes näkymättömästi [Orfali & Harkey, 98, s 281]. Kysymys siis herääkin: jos Java jo itsessään toteuttaa hajautetut oliot, mihin tarvitaan Corban kaltaista ulkopuolista oliovälitintä.

Myös OMG on tutkinut asiaa, ja julkaisemassaan artikkelissa David Curtis [97] esittääkin Corban käyttöönottoa monestakin syystä. Yksi syy on, että maailmasta löytyy miljardien dollarien arvoisia tietojärjestelmiä, joiden

kaikkien korvaaminen Javalla toteutetuilla järjestelmillä ei ole järkevää. Corba mahdollistaa näiden järjestelmien lähes saumattoman liittämisen uusiin järjestelmiin.

Myös Javan lähes täydellinen siirrettävyys ympäristöjen välillä on osaksi saavutettu suorituskyvyn kustannuksella. Virtuaalikone haukkaa aina osansa resursseista ja lisäksi ohjelmoijan kannalta erinomainen muistinhallintakaan ei tule ilmaiseksi. Raskaiden palvelinohjelmistojen toteuttaminen Javalla ei siis ole missään mielessä mielekästä.

Viime aikoina tosin suorituskykyä ei ole enää pidetty kovin vakavana ongelmana, sillä virtuaalikoneiden nopeutta on saatu nostettua huimasti ja JIT-kääntäjät (*Just in Time*) parantavat nopeutta entisestään. Tässä on päästy jopa niin pitkälle, että markkinoille on alkanut ilmestyä vakavasti otettavia Javalla toteutettuja palvelintuotteita. Suorituskyky kohoaa entisestään myös niin kutsutun 'Hotspot' –teknologian avulla, jossa ohjelmien kriittisiä kohtia tehostetaan käännösvaiheessa. Erityisesti laskentaoperaatioita ja palvelinkoneissa tärkeätä säikeiden käsittelyä on parannettu [Javasoft, 99].

Yksi syy on myös se, että tietotekniikan saralla ainoa pysyvä asia vaikuttaa olevan nopea muutos; parhaatkin järjestelmät muuttuvat ajan mittaan vanhanaikaisiksi. Java voi tällä hetkellä olla ohjelmointikielten viimeisintä huutoa, mutta se ei varmastikaan jää maailman viimeiseksi ohjelmointikieleksi. Nykyisistä muodikkaista ja uusia tuulia mukanaan tuovista Java-sovelluksista tulee ajan mittaan vain uusia jäännejärjestelmiä, joiden liittäminen vielä uudempiin järjestelmiin voi aiheuttaa ongelmia.

Lyhyenä listana Curtisin esittämät syyt siis ovat:

1. Jäännejärjestelmät
2. Suorituskyky
3. Java ei ole ikuinen

Nämä ongelmat eivät haittaa, mikäli päätetään käyttää Corbaa.

Corban voimahan on siinä, että se on ainoastaan määritelmä niistä yhteisistä palveluista ja rajapinnoista, joita Corba-toteutukseen tulee kuulua. Corba ei siis määrittele itse toteutustapaa millään lailla, eikä se siksi vanhene

tavallisten sovellusten tapaan. Olennaisena tekijänä on siis sopeutuminen uusiin olosuhteisiin, ja Corban ohjelmointikieli- ja ympäristöriippumattomuushan pyrkii juuri tähän.

RMI:n pahin heikkous piilee sen suurimmassa vahvuudessa: se on niin saumattomasti osa Javaa, että sen liittymät muihin ohjelmointikieliin ovat lähes mahdottomia toteuttaa. Javan versiossa 1.2 eli *Java Platform 2*:ssa tarjotaan kuitenkin erillisenä osana IIOP-liittymää, joten Corba on nykyään jo osa Java-kieltä. Jatkossa IIOP tullaan liittämään kiinteäksi osaksi Javaa. Näin Java-ohjelmoijat voivat jatkaa työtään Javalla ja kutsua tai tarjota kutsuttavaksi IDL-kielellä määriteltyjä rajapintoja.

Viimeisimpänä lisäyksenä Javan ORB:iin on tullut parametrien välittäminen arvoina, joten *inout*- ja *out*-parametreja voi nykyään käyttää.

2.2.2. Miten Java hyödyttää Corbaa

Corba on siis ohjelmointikieli- ja ympäristöriippumaton välittäjäohjelmisto. Miksi sitä pitäisi käyttää nimenomaan Javan kanssa, jota kaikki ohjelmoijat eivät välttämättä osaa. Miksi ei pysyttäisi edelleen C++:ssa, joka on varmasti suurimmalla osalla alan ihmisistä jo valmiiksi hallussa?

Robert Orfali ja Dan Harkey [Orfali & Harkey, 98, s.42] ovat listanneet niitä Javan piirteitä, jotka puolustavat sen käyttöä Corban kanssa. Tässä niistä tärkeimmät:

- Javan avulla Corba voi hajauttaa toimintoja
- Java on tekemässä Corbasta yleiskäyttöisen WWW:ssä
- Java tekee ohjelmien jakelusta laajoissa järjestelmissä helpompaa
- Java on hyvä kieli Corba-olioiden kirjoittamiseen

Toimintojen hajauttaminen on mahdollista, koska Java tukee liikkuvaa koodia, jonka avulla älykkyyttä voidaan siirtää sinne, missä sitä eniten tarvitaan. Tämä hajauttaminen voidaan tehdä jopa ajon aikana.

Javan vaikutus Corban yleistymiseen WWW:ssä johtuu muun muassa siitä, että Netscape on ilmaissut aikeensa sisällyttää ostamansa Visigenicin VisiBroker-ORBin kaikkiin tuleviin verkkoselaimiinsa ja palvelimiinsa. Näin

verkkosivuilla ajettavat 'appletit' voivat toimia missä tahansa käyttöjärjestelmässä ja myös käyttää Corba-palveluja ilman minkäänlaista asentamista.

Ohjelmien jakelu on helppoa, koska Java-ohjelmia voidaan hallita keskitetysti yhdeltä palvelimelta. Sovellukset päivitetään tälle palvelimelle, ja asiakasohjelmat itse huomaavat, milloin uuden version hakeminen on tarpeen. Näin säästetään ylläpitokustannuksissa.

Javan vahvuus kielenä johtuu siihen alusta asti sisäänrakennetuista ominaisuuksista kuten kieleen rakennetut säikeet, roskankeruu ja poikkeusten käsittely. Nämä helpottavat varmatoimisten komponenttien kirjoittamista. Javan oliomallikin muistuttaa Corbaa; molemmissa on rajapinnan käsite, joka erotetaan toteutuksesta. Kaiken kaikkiaan Java on tällä hetkellä kielenä ihanteellisin Corban ohjelmointiin.

2.3. Corba ja kilpailijat

Kuten aiemmin on mainittukin, Corba ei ole ainoa mahdollisuus hajautettujen järjestelmien toteuttamiseen Internetissä. Vaihtoehtoja ovat mm. TCP/IP, HTTP & CGI, RMI, Caffeine ja DCOM. Näistä oliomenetelmiin perustuvat RMI, Caffeine ja DCOM joita käsitellään seuraavassa lyhyesti.

2.3.1. RMI (Remote Method Invocation)

RMI on osa Javaa alkaen JDK 1.1:stä. Se on suunniteltu tukemaan metodikutsuja Java-virtuaalikoneiden välillä huomaamattomasti. Näin hajautetuista olioista tulee kiinteä osa ohjelmointikieltä. [Orfali & Harkey, 98, s. 281]

RMI:n käyttö muistuttaa hyvin paljon Corban käyttöä. Suurin ero on, että RMI:ssä luokat kuvataan yksinkertaisesti Java-rajapintoina (*interface*) eikä erillistä kuvauskieltä siis tarvita.

Käytettäessä RMI:tä kirjoitetaan luokan määrittely ensin Java-rajapintana, joka perii Javan standardikirjastoista löytyvän rajapinnan Remote. Tämän jälkeen kirjoitetaan rajapinnan toteuttava luokka, joka useimmissa tapauksissa peritään valmiista perusetäoliosta. Nämä luokat käännetään ensin Java-kääntäjällä ja saaduista binääritiedostoista tuotetaan *rmic*-

kääntäjällä asiakastynkä ja oliorunko, jotka vastaavat Corbasta tuttuja vastineitaan.

Olioiden löytäminen hajautetusta järjestelmästä tapahtuu RMI-järjestelmässä löytyvän rekisterin kautta. Tästä rekisteristä asiakassovellukset voivat löytää oliot jos ne vain tietävät oliota ajavan palvelimen ja olion itsensä nimen.

Mikäli kehittäjä haluaa käyttää hajautettuja olioita heterogeenisessä ympäristössä (eli myös muilla kielillä kuin Javalla), on Corban käyttö järkevin vaihtoehto. Luvussa 2.2 esitettyjen syiden takia kannattaa RMI:n käyttöä harkita tarkkaan myös sellaisissa projekteissa, joissa nykyisin käytetään vain Javaa. RMI sopii parhaiten pieniin ja lyhytikäisiin projekteihin.

2.3.2. Caffeine

Caffeine on Netscapen ja Visigenicin puhtaalla Javalla toteuttama IIOP-ratkaisu. Caffeine toimitetaan Netscapen Enterprise Server 3.0:n mukana. Se on vahva osoitus siitä, että Netscape on koko painoarvollaan Corban takana.

Caffeine koostuu kolmesta osasta: 1) *java2iiop*-kääntäjä, joka luo java-rajapinnoista IIOP-käyttöön soveltuvat asiakastyngät ja palvelinrungot, 2) *java2idl*-kääntäjä, joka kääntää java-koodin IDL-määritelmäksi sekä 3) URL:eihin perustuva nimeämispalvelu Web Naming Service, jonka avulla olioviittaus voidaan yksikäsitteisesti liittää URLiin. [Netscape, 97]

Caffeinea käyttäen kehittäjä luo ensin olion rajapinnan ja kirjoittaa sen Java-rajapintana. Tämä rajapinta käännetään ja saatu luokka käsitellään *java2iiop*-ohjelmalla. Näin saadaan asiakaspuolen tynkäluokka, palvelinolion runko ja kokoelmaluokka, joka sisältää palvelinolioista ulos tulevat parametrit.

Caffeine osaa myös tuottaa Java-rajapinnoista IDL-kielisen kuvauksen. Näin Javaan tottunut ohjelmoija voi unohtaa kokonaan Corban ja kuvitella tekevänsä pelkästään Java-sovellusta. C++:aa tai Cobolia käyttävä kehittäjä voi sitten jälkeenpäin generoida jo valmiista rajapinnoista IDL-kuvauksen ja

käyttää tätä ohjelmoidakseen valmiita palvelinolioita hyödyntävän asiakassovelluksen.

Caffeine saattaa hyvinkin olla paras ratkaisu kehittäjälle, joka haluaa tuottaa hajautettuja järjestelmiä Javalla tarjoten kuitenkin Corban käyttäjille helpon tavan päästä järjestelmään.

Caffeinen puute on, että RMI:n tavoin se toimii vain Java-kielen kanssa. Tosin IIOP ratkaisee tämän ongelman molempien osalta.

2.3.3. DCOM / ActiveX

DCOM on Microsoftin standardi hajautettujen olioiden toteuttamiseen. Jos Corba on hajautettujen olioiden maailman hallitseva standardi, on DCOM ehdottomasti toinen tärkeä, ja käytännössä ainakin vielä kaikkein käytetyin, standardi [Orfali & Harkey, 98, s. 331].

Vuonna 1990 Microsoft esitteli OLE-teknologian, jonka avulla sovellukset saattoivat jakaa tietoa keskenään. OLE on rakennettu eräänlaisen oliopyyntövälittäjän, COMin (*Component Object Model*) päälle. [Orfali *et al.*, 96, s. 283, 285]

DCOM (*Distributed COM*) on hajautettuihin ympäristöihin laajennettu versio COMista, ja se julkaistiin loppuvuodesta 1996, mistä alkaen se on ollut saatavissa Windows 95 ja Windows NT 4.0 -käyttöjärjestelmiin.

DCOMin tärkein etu on Microsoft, joka pelkällä omalla painoarvollaan tekee kaikesta julkaisemastaan huomionarvoista. Gartner Groupin ohjelmistoteknologian varatoimitusjohtaja Roy Schulte onkin valmis väittämään, että kun hajautettujen olioiden standardit ovat käyneet taistelunsa, jää Microsoft voittajana kentälle [Edwards, 97, s. 47]. Toisaalta Robert Orfali ja Dan Harkey uskovat Corban selkeään tekniseen ylivoimaan, ja povaavat siitä hallitsevaa standardia [Orfali & Harkey, 98, s. 371].

Muita hyviä syitä DCOMin käyttöön ovat erittäin hyvät kehitysvälineet ja OLEa ennestään tunteville myös tuttuus; ohjelmien kirjoittaminen ei eroa juurikaan vanhan COMin mallista.

Mutta DCOMilla on myös ongelmansa. OMG:n julkaisemassa artikkelissa on mainittu mm. seuraavat syyt, joiden takia Corba on DCOMiin verrattuna ylivertainen vaihtoehto [OMG, 97]:

Arkkitehtuuri, määrittely ja spesifikaatio

ActiveX ja DCOM perustuvat samaan teknologiaan, jolla Windows-maailmassa alunperin toteutettiin kopiointi- ja liittämistoiminnot ja joka tuolloin tunnettiin nimellä OLE. Sittemmin OLE on kokenut muodon- ja nimenmuutoksia ja sitä on jatkuvasti muokattu sopimaan kulloiseenkin Microsoftin strategiaan. On itse asiassa hyvin vaikea edes määritellä, mitä ActiveX sisältää, sillä Microsoftin mukaan ”ActiveX edustaa kieliriippumatonta rajapintaa, joka perustuu ActiveX-komponenttiteknologiaan”. Toisaalta käyttökelpoisten ActiveX-komponenttien tekeminen edellyttää 32-bittisen Windows-APIn käyttöä, joten sekin voitaisiin ymmärtää osaksi ActiveX:ää.

Toisaalta Corba on yksittäisistä valmistajista riippumaton standardi, jonka kehitys tapahtuu julkisena prosessina, johon mikä tahansa OMG:n jäsenyritys tai –yhteisö saa ottaa osaa. Millään yksittäisellä valmistajalla ei ole mahdollisuutta sitoa Corban standardia liikaa itseensä ja kuka tahansa saa korvauksetta toteuttaa standardin mukaisia tuotteita. Kaikki spesifikaatiot ovat vapaasti saatavissa esimerkiksi OMG:n WWW-sivuilta.

Tuki eri ympäristöille

ActiveX/DCOM on toteutettu ainoastaan Windows-ympäristöön, tarkemmin sanottuna Windows 95:een ja NT:hen sekä niiden seuraajiin. Microsoft on palkannut alihankkijoita toteuttamaan DCOMin myös Unix- ja MVS-ympäristöihin, mutta niiden julkaisupäivistä ei ole tietoa. Lisäksi Microsoftin selkeä kanta on, että he varmistavat sen, että ActiveX toimii parhaiten Windowsissa.

Yksi Corban tärkeimmistä suunnittelulähtökohdista on aina ollut ympäristöriippumattomuus. Riippuvuuksia minkään yksittäisen laitteiston tai ohjelmiston erikoisuuksista on haluttu välttää. Tänä päivänä Corba-tukea ei

ole ainoastaan useammalle käyttöjärjestelmälle kuin ActiveX-tukea; Corba-toteutus löytyy jopa useammalle Microsoftin käyttöjärjestelmälle, sillä siinä missä ActiveX toimii vain 32-bittisissä Windowseissa, on Corba toteutettu myös Windows 3.1:lle ja MS-DOSille.

Kieliriippumattomuus

DCOMin ohjelmointimalli on kiinteässä yhteydessä C- ja C++-kieliin. Tämän huomaa varsinkin C-tyylisistä muistiosoittimista, joita sovellukset lähettelevät toisilleen. Kuitenkaan esimerkiksi COBOLin ja Javan kaltaiset kielet eivät tunne osoittimen käsitettä.

Corban kieliriippumaton lähestymistapa taas on suunniteltu toimimaan mahdollisimman useiden kielten kanssa. OMG on määritellyt kielisidonnat C:lle, C++:lle, Adalle ja Smalltalkille. COBOL- ja Java-sidonnat on jo toteutettu ja niiden liittymiä standardoidaan paraikaa. Näiden lisäksi yksittäiset toimittajat ovat tehneet omat määritelmänsä sidonnoista Visual Basiciin ja Fortraniin.

Ympäristöjen kypsyneisyys

COM on historiansa aikana joutunut kärsimään jatkuvista spesifikaatioiden ja koodin muutoksista. Kolme kuukautta DCOMin julkistamisen jälkeen sen avaindokumentit olivat vielä keskeneräisiä ja julkaistu dokumentointi sisälsi aukkokohtia.

Corban määritelmät sitä vastoin julkaistiin vuoden 1991 alkupuolella teollisuudenalan laajalla tuella ja toteutuksia alkoi ilmestyä saman vuoden maaliskuussa. Standardin versio 2.0 julkaistiin joulukuussa 1994. Se oli yhtälailla tuettu ja toteutukset ilmestyivät nopeasti.

DCOMin vasta saapuessa loppukäyttäjien ulottuville, on Corba käytössä kokeiltu ja kypsä teknologia, jota on käytetty suurissa yrityksissä jo yli kuuden vuoden ajan.

Turvallisuus

ActiveX-komponenttien turvallisuusriskit ovat herättäneet paljon kritiikkiä. ActiveX-komponentit toimivat kuten normaalit sovellukset eikä niiden oikeuksia voida rajoittaa millään lailla. Siksi ne voivat tuhota levyille tallennettua tietoa tai lähettää arkaluontoisia tietoja verkon yli asiaankuulumattomien käsiteltäväksi. Turvallisuus hajautetuissa ympäristöissä koostuu monesta muustakin seikasta. Arkaluontoiset tiedot tulee voida lähettää koodattuna, järjestelmään tai sen osiin pääsemistä tulee voida kontrolloida joissakin tapauksissa täytyy voida kiistämättömästi todistaa, että joku järjestelmän käyttäjä on todella tehnyt mitä on tehnyt. Viimeisin ominaisuus on välttämätön esimerkiksi rahaliikenteessä, jotta rahansiirron osapuolet eivät voi kieltää osuuttaan asiaan.

ActiveX ei toistaiseksi tue mitään edellä mainituista turvallisuusominaisuuksista kun taas Corbassa kaikki nämä on määritelty ja ne on jopa toteutettu joissakin myytävissä tuotteissa.

Skaalautuvuus

ActiveX:n suurimmat ongelmat ilmenevät, kun järjestelmiä aletaan kasvattaa mielivaltaisen suuriksi. Tällöin yhdelle koneelle alun perin suunniteltu järjestelmä muuttuu horjuvaksi.

DCOM-ympäristössä jokaiselle oliolle pidetään yllä ns. viitelaskuria, joka kertoo, kuinka moni asiakassovellus käyttää kutakin oliota. Kun viitelaskuri laskee nollaan, ei oliota enää tarvita ja se voidaan tuhota. Ohjelmoijan kannalta valitettavasti viitelaskurin käsittely on hänen tehtävänsä. Jos jonkin olion viitelaskuria kasvatetaan yhdenkin kerran useammin, kuin pitäisi, voi olio jäädä ikuisesti kuormittamaan järjestelmää. Toisaalta taas yksikin ylimääräinen viitelaskurin vähennys voi hävittää vielä tarvittavan olion kokonaan, mikä on vielä kiusallisempaa kuin tarpeettomat haamuoliot.

Vaikka ohjelmakoodissa viittauksien ylläpito olisikin hoidettu virheettömästi, voivat verkossa usein esiintyvät yhteysongelmat aiheuttaa viittauksen lisäys- tai poisto-operaation katoamisen.

Koska viittauslaskuri on niin epävarma, käytetään DCOMissa varajärjestelmänä elossapitokutsuja (*keepalive messages*), joita DCOM-kirjallisuudessa kutsutaan *'pingeiksi'*. Jokaisen asiakasolion oletetaan lähettävän elossapitokutsuja niille olioille, joiden käyttämisestä voidaan vielä jossain vaiheessa tulevaisuudessa olla kiinnostuneita. Kutsujen välinen aika on kaksi minuuttia ja jos jotakin oliota ei kuuden minuutin aikana ole kutsuttu kertaakaan, se saa tuhota itsensä riippumatta viittauslaskurin lukemasta. Jos palvelin siis joutuisi verkkohäiriön vuoksi eroon asiakaskoneista kuuden minuutin ajaksi, se katoaa kokonaan. Tämä ei OMG:n vertailun mukaan ole oikea ratkaisu maailmanlaajuisen tietoverkon arkkitehtuuriksi.

Kaikki nämä ongelmat johtuvat siitä, että DCOM on pohjimmiltaan yhdessä koneessa ajettavien sovellusten kommunikointiväylä, eikä sitä ole tarkoitettu ratkaisuksi verkkoympäristöihin. Corba puolestaan on alusta lähtien suunniteltu toimimaan verkkoympäristössä, joten sen toteutuksissa on otettu nämä ongelmat huomioon koko ajan.

Vaikka edellä esitetyt ongelmat onkin lueteltu OMG:n julkaisemassa artikkelissa joka ei arvatenkaan ole täysin pyyteettömästi kirjoitettu, ovat ne kuitenkin olemassa olevia ongelmia, jotka täytyy ottaa huomioon joko tehtäessä ActiveX-sovelluksia tai valittaessa toteutusvälinettä hajautetun järjestelmän pohjaksi.

Toinen vertailu

Tammikuun 1998 C++ Report -lehdessä [Chung *et al.*, 98] on vertailtu Corbaa ja DCOMia kolmella tasolla: perusohjelmointiarkkitehtuurina, etäohjelmointiarkkitehtuurina ja verkkoyhteysarkkitehtuurina.

1. Perusohjelmointiarkkitehtuuri

   Tällä tasolla suurin ero on siinä, miten rajapinta määritellään.

   Corban oliomalli muistuttaa enemmän klassista olioparadigmaa, jossa määritellään luokka, ja yksittäisiä luokan ilmentymiä kutsutaan olioiksi. DCOMissa taas olio on vain kokoelma proseduureja tai funktioita, joita voidaan kutsua tämän olion kautta. DCOM-oliolla ei siis voi olla tilaa.

DCOMissa oliot luodaan tyypillisesti kutsumalla ensin COMin *CoCreateInstance*-funktiota antaen parametrina luotavan olion yksilöivä ID-tunnus. Olion kahva saadaan asiakassovelluksessa käytettäväksi kutsumalla *IClassFactory*n eli luokan tehtaan *CreateInstance*-metodia, joka ei välttämättä palauta aina uutta oliota vaan voi joka kerralla antaa kahvan yhteen ja samaan olioon.

Corba-määritelmän mukaan paras tapa saada olioviite on käyttää standardin mukaista *Trader*-palvelua, mutta tämä vaihtelee eri toteutusten välillä. Hyvin yleinen tapa on käyttää BOA:n (*Basic Object Adapter*, oliosovitin) metodeita. Olioviite voidaan tallettaa merkkijonona, joka saadaan Corban standardimetodilla *object_to_string*.

Toinen tärkeä ero on poikkeusten käsittely. Corba tukee suoraan C++:n poikkeuksia, minkä lisäksi siinä on oma poikkeustenkäsittelymääritelmä esimerkiksi C-ohjelmia varten. DCOM taas edellyttää, että jokainen funktio palauttaa 32-bittisen virhekoodin *HRESULT*. Tosin jotkin kehitysvälineet voivat luoda paljaan DCOMin päälle oman kerroksen, jolla virhekoodi voidaan muuttaa poikkeukseksi, jota voidaan käsitellä tavalliseen tapaan asiakassovelluksessa.

2. Etäohjelmointiarkkitehtuuri

   Keskimmäinen kerros koostuu siitä infrastruktuurista, joka mahdollistaa asiakkaan ja palvelimen ohjelmoinnin kiinnittämättä lainkaan huomiota siihen, että ne toimivat eri prosesseina tai mahdollisesti jopa eri koneissa.

   Tärkeimpiä eroja Corban ja DCOMin välillä tässä kerroksessa ovat palvelinolioiden rekisteröinti ja olion tyngän ja rungon luomisajankohdat.

   Eri sovellusten välisen kommunikoinnin mahdollistamiseksi täytyy välitettävä tieto järjestellä (marshal ja unmarshal) sellaiseen muotoon, että se on siirrettävissä välitysmekanismeilla osoiteavaruuksien välillä ja purettavissa vastaanottavassa sovelluksessa. Järjestelyssä metodin parametrit tai palautusarvot pakataan jonkin standardin mukaiseen siirtomuotoon.

DCOMissa tietojen järjestelystä käytetään nimitystä *standard marshaling*. Käyttäjän on mahdollista ohittaa tämä tavallisesti käytetty järjestelytapa luomalla oma *custom marshaling* -mekanismi, jonka avulla voidaan pitää asiakassovelluksessa omaa välimuistia tai parantaa virheensietoa.

DCOMissa olion aktivointi käynnistetään kutsumalla CoCreateInstance-funktioita, ja varsinaisen työn tekee *Service Control Manager (SCM)* eli *palveluiden hallinta.* SCM tarkistaa, onko halutun olion ID-tunnus jo rekisteröity ja jos ei, etsii Windowsin registrystä palvelimen polkunimen ja käynnistää palvelinsovelluksen. Palvelinsovellus rekisteröi kaikki tarjoamansa oliot ja SCM luo halutun olion tehdasta käyttäen.

Kun järjestelmä saa käytettäväksi osoittimen olioon, se luo oliotyngän (joka vastaa Corban runkoa) ja järjestelee osoittimen asiakkaalle tarjottavaan muotoon. SCM palauttaa osoittimen asiakkaan päähän ja COM-järjestelmä luo oliolle välittäjän (joka vastaan Corban asiakastynkää). Välittäjä järjestelee osoittimen asiakassovelluksen käyttämään muotoon ja liittää tähän RPC-kanavaolion (Remote Procedure Call). Lopuksi COM-järjestelmä palauttaa asiakassovellukselle osoittimen pyydettyyn olioon.

Chungin ja työryhmän käyttämässä Orbixissa kutsutaan halutun olion staattista metodia *_bind*, jolloin oliotynkä välittää pyynnön ORB:lle. (Mikäli oliota on jo käytetty, tietää asiakastynkä tästä ja osaa kutsua olion runkoa ilman ORBin apua.) ORB käynnistää toteutustietokantaa apuna käyttäen palvelimen. Palvelin luo tarvittavat oliot ja ilmoittaa BOAlle tarjolla olevista olioista. Palvelinpäässä luokan rakennin (*constructor*) luo myös runkoluokan ilmentymän. Palauttaessaan olioviitettä ORB luo instanssin halutun olion tyngästä ja viite tähän palautetaan asiakassovellukselle.

3. Verkkoyhteysarkkitehtuuri

   Corba-standardi ei määrää, miten asiakas ja palvelin kommunikoivat keskenään yhden ORB:n sisällä. Eri ORB-tuotteiden väliseen kommunikointiin on sen sijaan määritelty General Inter-ORB Protocol (GIOP) eli

yleinen ORBien välinen protokolla. GIOP:n toteutus TCP/IP-yhteyksillä tunnetaan usein käytetyllä nimellään IIOP.

DCOM perustuu OSF:n DCE-määritelmään, joskin siihen on tehty joitain lisäyksiä. Näistä yksi on aiemminkin mainitut pingit. Ping-viestit lähetetään koneittain ja ne voidaan upottaa normaalien viestien sekaan. Ping-toiminnallisuus voidaan myös kytkeä pois käytöstä.

Kuten kohdassa 1 sanotaan, DCOM-oliot eivät ole olioita klassisen olio-ohjelmoinnin tapaan. Microsoft on kuitenkin kiertänyt tämän ongelman käyttämällä yhteystunnisteita (*moniker*). Yhteystunnisteet ovat olioita, jotka toimivat eräänlaisina aliaksina jollekin toiselle oliolle, joka puolestaan voi sisältää tilatietoa. Näin ei tavallaan niinkään kutsuta olion metodia vaan pikemminkin annetaan olio yhtenä funktion parametrina.

Vaikka Corballa ja DCOMilla onkin paljon eroavuuksia, ovat ne molemmat kuitenkin tahoillaan paikkaansa puolustavia ratkaisuja. DCOMin valtti on hyvät kehitysympäristöt, joihin senkin ominaisuudet on integroitu. Corban etu taas on toimiminen lähes kaikilla käytössä olevilla laitealustoilla ja laaja tuki eri valmistajilta. Huomattava on myös, että DCOM on pohjimmiltaan hyvin C++-keskeinen ympäristö kun taas Corba tukee lähes kaikkia yleisemmin käytettyjä ohjelmointikieliä.

Tärkein ero on se, että Corba on standardi, jonka mukaan kuka tahansa valmistaja saa tuottaa omat toteutuksensa ja DCOM puolestaan on yhden valmistajan toteutus, jonka mukaan standardi muotoutuu. Maailmojasyleilevään tyyliinsä OMG on luomassa yhteyttä Corban ja DCOMin välille välineenään IIOP. Kun tämäkin määritelmä on valmis, ei käyttäjän ole pakko tehdä näidenkään järjestelmien välistä joko-tai-valintaa; voi valita molemmat.

Jos valinta kuitenkin tehdään, kannattaa ottaa huomioon, että DCOM on vain yksi toteutus, jolla ei ole kilpailijaa. Microsoft saattaa tehdä toteutukseen ja rajapintoihin muutoksia, jotka taas voivat johtaa muutoksiin ohjelmakoodissa. Pahimmassa tapauksessa toiminta muuttuu epävarmemmaksi kuin

aiemmissa versioissa tai toiminnallisuus muuttuu, kuten on tapahtunut esimerkiksi MFC-luokkakirjaston kanssa.

Lukuisat Corba-toteutukset taas kilpailevat keskenään ja siksi niiden kehittäjillä on jonkinasteisia suorituspaineita. Lisäksi mahdolliset muutokset ohjelmointikielten sidontoihin joutuvat käymään läpi sellaisen byrokraattisen pyörityksen, että niitä ei luultavasti kovin usein tapahdu. Ja mikäli tapahtuu, ainakin muutokset näkyvät määritelmissä ennen kuin toteutuksissa.

Mikäli joutuisin tekemään valinnan näiden kahden vaihtoehdon välillä, valitsisin mieluummin Corban, koska se on aidosti oliopohjainen järjestelmä; joskus jopa tuskastuttavan oliopohjainen, kuten myöhemmin kerron. Ohjelmointi nykyään perustuu niin vahvasti olioihin, että funktiorajapinta tuntuu vanhanaikaiselta. Lisäksi DCOM-ohjelmointi vaikuttaa esimerkiksi Orfalin ja Harkeyn [98] kirjassa esitetyn esimerkin perusteella aivan liian monimutkaiselta.

2.4. Välineet pähkinänkuoressa

Robert Orfali ja Dan Harkey [98, s. 375] ovat esittäneet kirjassaan yhdessä taulukossa vertailun eri hajautettujen järjestelmien toteutusvälineiden välillä; seuraavassa on taulukon sisältö Corban, DCOMin ja RMI:n osalta. Caffeinea ei taulukossa ole, koska se on vain yksi Corba ORB.

Taulukko 1 : Corban ja kilpailijoiden vertailu

Piirre	**CORBA / IIOP**	**DCOM**	**RMI**
Tuki eri käyttöjärjestelmille	★★★★	★★	★★★★
100% Java-toteutus	★★★★	★	★★★★
Etämetodien kutsut	★★★★	★★★	★★★
Tila säilyy metodikutsujen välillä	★★★★	★★★	★★★
Dynaaminen olioiden löytäminen	★★★★	★★★	★★
Dynaamiset metodikutsut	★★★★	★★★★	★
Yhteyksien suojaus	★★★★	★★★★	★★★★
Pysyvät olioviitteet	★★★★	★	☆
URL-pohjainen nimeäminen	★★★★	★★	★★
Olioiden käyttäminen eri kielillä	★★★★	★★★★	☆
Kieliriippumaton yhteysprotokolla	★★★★	★★★★	☆
Skaalautuvuus	★★★★	★★	★
Avoin standardi	★★★★	★★	★★

Taulukossa ei ole esitetty niitä piirteitä, joista nämä kolme saivat saman arvosanan. Tällaisia piirteitä olivat esimerkiksi abstraktiotaso, tuki parametrien tyypeille ja Java-integraation saumattomuus. Corban ja RMI:n kohdalla tämä saumattomuus on helposti ymmärrettävissä, DCOMin kohdalla taas se johtuu hyvistä kehitysvälineistä, jotka tarjoavat kehittäjälle illuusion siitä.

Kaikkien kohtien merkitys ei ole aivan itsestään selvää. Esitetyt ominaisuudet tarkoittavat seuraavaa:

- **Tuki eri käyttöjärjestelmille**: Corba-toteutus löytyy kaikista tärkeimmistä käyttöjärjestelmistä, RMI-toteutus taas löytyy niistä käyttöjärjestelmistä, joille on tehty Java-toteutus. DCOM toimii täysin vain Windows NT- ja Windows 95-käyttöjärjestelmissä. DCOMin heikot pisteet tulevat siitä, että sen palvelinpuolen ympäristövalikoima ei ole kovin vahva.
- **100% Java-toteutus**: Puhtaalla Javalla toteutettuja Corba-ORBeja on olemassa, DCOM taas on kirjoitettu C:tä ja C++:aa käyttäen. Java-adapterit

löytyvät Microsoftin omasta virtuaalikoneesta, mutta muilla Java-järjestelmillä DCOMia ei voi käyttää.

- **Etämetodien kutsut**: Sekä Corba, DCOM että RMI tukevat kyllä etämetodikutsuja, mutta Corba saa korkeammat pisteet, koska se tukee yksilöiviä olioviitteitä, joiden avulla oliot voidaan aktivoida tarvittaessa.
- **Tilan säilyminen metodikutsujen välillä**: Toisin kuin DCE-ympäristöissä tai HTTP / CGI-toteutuksissa, ORBeissa metodikutsut voidaan osoittaa tietyille olioille. Corban korkeammat pisteet johtuvat jälleen yksilöivistä viitteistä, joiden avulla tila voidaan säilyttää jopa eri yhteyskertojen välillä.
- **Dynaaminen olioiden löytäminen**: Sekä Corba että DCOM mahdollistavat rajapintojen löytämisen ajon aikana ilman että niitä olisi kiinteästi määritelty ohjelmakoodissa. Paikallisessa rajapintatietokannassa varastoitujen rajapintojen lisäksi Corba mahdollistaa myös eri ORBeissa sijaitsevien rajapintojen löytämisen IIOP:n avulla. DCOM tukee vain paikallisesti varastoitujen rajapintojen hakemista.
- **Dynaamiset metodikutsut**: Vain Corba ja DCOM mahdollistavat metodien dynaamisen kutsumisen. Teoriassa RMI mahdollistaa asiakastynkien lataamisen verkon ylitse ja metodien kutsumisen niiden avulla. Ratkaisu on kuitenkin hyvin kömpelö.
- **Yhteyksien suojaus**: DCOMin turvaominaisuudet ovat pohjimmiltaan samat kuin Windows NT:ssä, tulevaisuudessa aiotaan lisätä DCE:n tapainen yhteyksien suojaus. RMI:n yhteyksien suojaus perustuu SSL:ään eli *Secure Socket Layeriin*. Corba taas määrittelee turvallisuuspalvelun; myös Corbassa yhteyksien suojaamiseen voidaan käyttää SSL:ää.
- **Pysyvät olioviitteet**: Corba tukee pysyviä olioita ja viittauksia niihin. DCOM-olioilla taas ei varsinaisesti ole tilaa lainkaan, mutta ne voidaan yhdistää tilansa säilyttäviin olioihin käyttämällä yhteystunnisteita. Javan versiossa 1.2 RMI tukee pysyviä olioviitteitä ja Corba-tyylistä olioiden aktivointia, mutta toistaiseksi kumpaakaan ei ole käytettävissä.

- **URL-pohjainen nimeäminen**: Internetissä ja intraneteissä voidaan hyötyä URL-pohjaisesta nimeämisestä. RMI tukee URL-viitteitä suoraan ja Corba Caffeinen avulla. DCOM taas tukee niitä yhteystunnisteiden avulla.
- **Olioiden käyttäminen eri kielillä**: Corba-määritelmään kuuluvat standardit korkean tason kielten sidonnoista IDL-kieleen. DCOM taas on binääritason protokolla, jota tuetaan mm. Microsoftin kehitysvälineissä; sidontoja eri kieliin ei kuitenkaan ole. RMI toimii ainoastaan Javalla.
- **Kieliriippumaton yhteysprotokolla**: Sekä DCOM että Corba mahdollistavat monenlaisen tiedon siirtämisen viestien sisällä. Corban kielisidonnat yhdistävät datan erikielisiin esitysmuotoihin. DCOM taas käyttää OLE-automaatiotyyppejä tiedon esittämiseen sovelluksille. RMI toimii ainoastaan Javalla, joskin IIOP-yhteydet tarjotaan tulevissa versioissa.
- **Skaalautuvuus**: IIOP määrittelee säännöt yhteyksien pitämiseen eri valmistajien Corba-ympäristöjen välillä, joten oliot voivat kommunikoida keskenään toteutuksista riippumatta. Turvallisuusominaisuuksia ja transaktioita varten eri toimialueiden välillä on myös määritelty protokollat. Rajapintatietokannat ja muut Corba-palvelut toimivat myös. Corban valttina onkin juuri löyhästi toisiinsa sidotut ORBit, jotka voivat kommunikoida ja jakaa tietoa keskenään.
- **Avoin standardi**: Maailmanlaajuisen alustan täytyy perustua avoimiin standardeihin kuten Corban tapauksessa on asia. DCOMia hallitsee Microsoft, joka antaa määritelmät standardointi-instanssina toimivalle Open Groupille. JavaSoft määrää Javan ja näin ollen myös RMI:n määritelmistä, jotka ISO voi myöhemmin ratifioida.

2.5. Yhteenveto

Vaikka Orfalia ja Harkeyta voidaankin syyttää tarkoitushakuisuudesta näiden kriteerien valinnassa, on Corba monessa tärkeässä suhteessa edellä muita. Tärkeimpiä näistä ovat ympäristöriippumattomuus ja skaalautuvuus.

Ympäristöriippumattomuus on tärkeää. Corballa pyritään ratkaisemaan muun muassa vanhojen ja uusien sovellusten yhdistämisen aiheuttamat ongelmat. Tällöin laitteistot, käyttöjärjestelmät ja toteutuskielet voivat olla mitä tahansa.

Skaalautuvuus taas on tärkeää, koska eniten ratkaisuja kaipaavat tahot ovat rahoituslaitoksia, vakuutusyhtiöitä ja muita suuryrityksiä, joilla mahdollisten käyttäjien määrä voi nousta jopa kymmeniin tuhansiin.

Skaalautuvuus on erittäin tärkeää myös ryhmäohjelmien toteuttamisessa, koska niiden käyttöarvo kasvaa sitä suuremmaksi, mitä useampia ihmisiä on mahdollista tavoittaa.

RMI:n kompastuskiveksi muodostuu lopulta vanhat järjestelmät. Hyvin monet nykyisin käytössä olevat palvelimet on toteutettu käyttäen C++-, C- tai jopa Cobol-kieltä. Näistä C on ainoa, jonka päälle voidaan rakentaa Java-kerros. Tällaisen kerroksen rakentaminen taas ei ole juurikaan yksinkertaisempaa kuin suosiolla käyttää Corbaa.

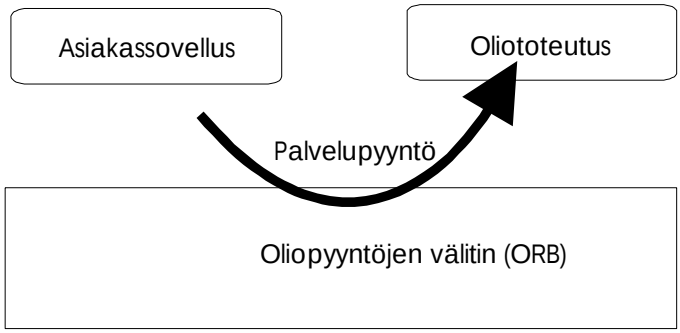
DCOMin heikkous on Windows. Vaikka Windows kaikkine eri versioineen onkin nykyään ylivoimaisesti suosituin työasemien käyttöjärjestelmä, on sen ongelmana epävarmuus ja siksi sen käyttäminen palvelinkoneissa on huomattavasti vähäisempää kuin Microsoft soisi. Voitaneen sanoa, että kaikki raskaammassa ja luotettavuutta vaativassa käytössä olevat palvelimet on toteutettu erilaisille Unix-alustoille, suosituimpana näistä ehkä Sunin Solaris. DCOM taas ei toimi Solariksessa, joten sen käyttäminen on mahdotonta.

Mielenkiintoista sinänsä on, että viime aikoina myös Linux on alkanut kelvata palvelinalustaksi; virallisesti vain harvoja kaupallisia tuotteita toimitetaan sille, mutta kysyntää on olemassa ja epävirallista kokeilua harjoitetaan useissakin yrityksissä. DCOMia Linuxille ei todennäköisesti ole luvassa.

Corba voidaan siis tuomita näiden välineiden välisessä taistossa voittajaksi, mutta seuraava kysymys onkin, mihin tätä taistoa ylipäätään tarvitaan. Eikö voitaisi jatkaa samaan malliin kuin tähänkin asti välittämättä uusista mahdollisuuksista. Tähän kysymykseen haetaan vastausta seuraavassa luvussa.

3. Corban rakenne

3.1. Yleiskatsaus



Kuva 1 : Palvelupyyntö kulkee asiakassovellukselta kohdeoliolle oliopyyntöjen välittäjän kautta

Tavallisissa asiakas-palvelin -sovelluksissa kehittäjä on itse toteuttanut yhteydenpidon. Tällöin asiakas kutsuu yleensä suoraan palvelinta, eikä välissä ole ylimääräisiä välikäsiä. Asiakassovelluksen on tällöin tiedettävä, mitä tietoliikenneprotokollaa käytetään sekä mikä on palvelinohjelmaa ajavan koneen verkko-osoite tai nimi.

Corba-maailmassa ei puhuta varsinaisesti asiakkaista ja palvelimista, vaan yksinkertaisesti olioista ja niiden rajapinnoista. Sovellukset voivat lähettää näille olioille metodikutsuja **oliopyyntöjen välittäjän** eli ORB:n (*Object Request Broker*) kautta. ORB paikantaa sovellukselta saamansa olioviitteen avulla halutun olion ja pitää huolen siitä, että mahdollisesti toisella ohjelmointikielellä annetut parametrit välitetään oliototeutukselle sen ymmärtämässä muodossa. Sitten ORB odottaa oliolta palautetta ja lähettää sen takaisin kutsuvalle sovellukselle.

Oliopyyntöjen välittämisen lisäksi Corba-toteutukset tarjoavat laajan joukon erilaisia palveluja, joiden avulla voidaan ottaa selvää tarjolla olevista oliorajapinnoista, valvoa niiden käyttöoikeuksia, huolehtia olioiden käyttöön

mahdollisesti liittyvästä laskutuksesta, tallettaa olioita pysyvään muistiin jne. Kaikki Corba-toteutukset eivät kuitenkaan toteuta kaikkia näitä palveluja ja joillakin toteutuksilla on omia erityispiirteitänsä, jotka eivät kuulu standardiin.

3.2. OMG IDL

Kaikissa hajautetuissa sovelluksissa asiakkaan ja palvelimen täytyy tietää, miten niiden tulee kommunikoida. Corbassa tähän tarkoitukseen käytetään määriteltäviä rajapintoja (*interface*). Rajapinnat määritellään erityisillä IDL-tiedostoilla.

IDL (*Interface Definition Language*) on OMG:n määrittelykieli, jolla olioiden ominaisuudet ja metodit voidaan määritellä tarkasti ja ohjelmointikielestä riippumatta. IDL:ää ei siis voi käyttää toiminnallisuuden koodaamiseen, vaan ainoastaan rajapintojen esittelyyn. Näin toiminnallisuuden ja rajapintojen ero saadaan tehtyä mahdollisimman selväksi. [Otte *et al*., 96, s. 2– 3]

IDL:ää käyttäen on mahdollista määritellä luokkia sekä niille ominaisuuksia ja metodeita. Ominaisuuksia varten voidaan määritellä omia tietotyyppejä, joista mainittakoon C++-tyyliset *struct* ja *enum* -rakenteet. Metodit voidaan määritellä heittämään poikkeuksia ja ne voivat saada ja palauttaa parametreja. IDL määrittelee myös erityisen asynkronisen *oneway*-metodin, jota kutsutaan, mutta jonka kutsumisen jälkeen asiakassovellus jatkaa suoraan suorittamistaan odottamatta palautetta oliolta.

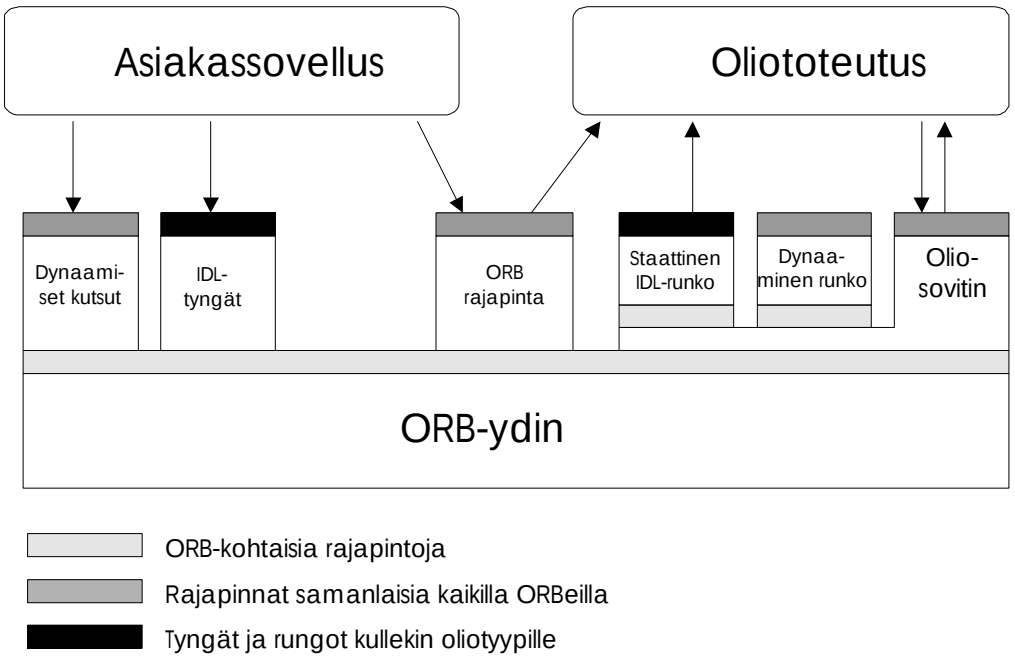
Oliokielten tapaan myös IDL:llä voidaan määritellä luokat perimään ominaisuuksia ja metodeita toisilta luokilta. Toisiinsa liittyvät luokat ja muuttujatyyppimäärittelyt voidaan ryhmitellä moduuleihin.

Pohjimmiltaan IDL:n rakenne on C++:n osajoukko, johon on lisätty avainsanoja hajautettujen järjestelmien toteuttamista varten, lisäksi IDL tukee C++-standardin mukaista esiprosessointia ja *pragma*-komentoja [Orfali & Harkey, 98, s. 5].

Jokaisen Corba-standardin mukaisen ORBin täytyy sisältää IDL-kääntäjä, jolla IDL-kielinen määritelmä käännetään varsinaiselle toteutus-

kielelle. Corban versiossa 2.0 oli määritelty sidonnat C:lle, C++:lle ja Smalltalkille. Versiossa 2.2 on mukana myös Java.

3.3. Oliopyyntöjen välittäjä (ORB)



Kuva 2 : Oliopyyntöjen välittäjän rajapintojen rakenne

Kuva 2 esittää yksittäisen oliopyyntövälittäjän rakenteen. ORB:n rajapinnat on esitetty sävytetyillä laatikoilla ja nuolet näyttävät, mihin suuntaan metodien kutsut liikkuvat. Olion metodia voidaan kutsua kahdella eri tavalla: joko käyttämällä olion IDL-tynkää eli juuri kohdeolion omaa rajapintaa tai käyttäen dynaamista kutsurajapintaa (DII, *Dynamic Invocation Interface*), jolloin rajapinta on aina sama riippumatta kutsuttavasta oliosta.

Oliototeutusta voidaan kutsua joko staattisen IDL-rungon tai dynaamisen rungon kautta. Oliototeutus taas voi kutsua ORB-rajapinnan tai oliosovittimen metodeita.

Olion rajapintamääritelmä voidaan esittää kahdella tavalla. Yksi tapa on tehdä siitä IDL-kielinen määritelmä. Toinen tapa, jota voidaan käyttää jo-

ko IDL-kielisen esityksen sijasta tai sen lisäksi on rajapintatietokanta. Rajapintatietokannassa sijaitsevien olioiden osat esitetään omina olioinaan, joihin on mahdollista päästä käsiksi ajon aikana. Sekä IDL-kieli että rajapintatietokanta ovat ilmaisuvoimaltaan yhtä vahvoja kaikissa ORB-toteutuksissa [CORBA, 98, s. 2–3].

Voidakseen esittää oliopyynnön, kutsuvan sovelluksen täytyy pitää hallussaan olioviitettä ja sen täytyy tuntea kutsuttavan olion tyyppi sekä operaatio, jota suoritetaan. Sekä dynaaminen että tynkää käyttävä kutsu ovat semanttisesti samanlaisia, eikä kutsun vastaanottava toteutus pysty päättelemään, kummalla tavalla kutsu esitettiin.

ORB etsii asianmukaisen toteutuksen, välittää parametrit ja siirtää kontrollin oliototeutukselle IDL-rungon tai dynaamisen rungon kautta. Pyyntöä täyttäessään oliototeutus voi pyytää ORB:lta joidenkin palveluiden apua. Kun pyyntö on suoritettu, palautetaan paluuarvot asiakkaalle.

Yleensä ORB:ia ei toteuteta yhtenä ainoana komponenttina vaan se määritellään joukkona rajapintoja. Nämä rajapinnat on jaettu kolmeen ryhmään:

1. Operaatiot, jotka ovat kaikille ORB-toteutuksille yhteisiä
2. Operaatiot, jotka ovat samanlaisia tietynlaisille olioille
3. Operaatiot, jotka ovat samanlaisia tietyn tyylisille oliototeutuksille

Joissakin tapauksissa on mahdollista, että asiakassovellus viittaa kahteen olioon, joiden toteutukset ovat eri ORBeissa. Tällöin on ORBien tehtävä erottaa nämä olioviitteet toisistaan; vastuuta ei ehdottomasti saa jättää asiakassovellukselle.

3.4. Oliosovitin (Object Adapter)

Oliosovittimet määrittelevät sen, kuinka oliot aktivoidaan. Jokainen palvelin voi tukea useaa oliosovitinta, joista jokaisella on oma käyttönsä. Oliototeutus voi päättää tarjoamiensa palveluiden mukaan sen, mitä oliosovitinta haluaa käyttää. Esimerkiksi oliotietokannan hallintajärjestelmä voisi haluta itse hallita kaikkia yksittäisiä tietoalkioitaan rekisteröimättä niitä kuitenkaan yksi-

tellen. Tällöin se voisi tarjota oman oliosovittimen tähän erikoistarkoitukseen. OMG ei kuitenkaan halua, että jokainen Corba-toteutus tarjoaa joukon erilaisia oliosovittimia, jotka suurimmaksi osaksi tarjoavat samanlaisia palveluita, joten Corba-spesifikaatiossa on määritelty yleisoliosovitin (Basic Object Adapter), jonka on oltava osa jokaista ORBia. [Orfali & Harkey, 98, ss. 393–394]

Yleisoliosovitin tarjoaa seuraavat palvelut:

- Toteutusvarasto, joka mahdollistaa toteutusten asentamisen ja niiden rekisteröimisen. Tänne voidaan myös tallentaa oliota kuvaavaa tietoa.
- Mekanismi olioviittausten tuottamiseen ja tulkitsemiseen, oliototeutusten käynnistämiseen ja sammuttamiseen ja metodien kutsumiseen ja niiden parametrien välittämiseen.
- Toteutusolioiden käynnistäminen ja sammuttaminen.
- Metodikutsut runkojen välityksellä.

Corban versiossa 2.2 on BOA korvattu *Portable Object Adapterilla* (POA), eli toteutusriippumattomalla oliosovittimella. Corba-määritelmässä sovittimelle on esitetty muun muassa seuraavat tehtävät [CORBA, 98]:

- Mahdollistaa palvelinolioiden toimimisen eri ORB-tuotteissa.
- Tukee olioiden pysyvää tunnistamista. Oliota käyttävän sovelluksen kannalta tämä tarkoittaa sitä, että vaikka palvelin sammutettaisiin ja käynnistettäisiin uudestaan, yhtä ja samaa oliota olisi mahdollista käyttää toistuvasti.
- Antaa yhden *palvelijan* hallita useita olion tunnisteita kerralla. Palvelija on käytännössä yksi palvelinkoneen prosessi, joka toteuttaa yhden tai useamman olion metodikutsut.
- Antaa ohjelmoijan rakentaa olioita, jotka periytyvät staattisista runkoluokista tai dynaamisesti luoduista toteutusrungoista.

3.5. Yleiset palvelut

Corban lisäksi OMG on määritellyt itse tai hyväksynyt joukon palveluita, jotka voivat olla osa ORB-toteutusta. Nämä määritelmät löytyvät *CORBA services*-dokumentista [OMG, 97/2]. Seuraavassa on lyhyt kuvaus dokumentissa esitellyistä palveluista.

Nimeämispalvelu (*Naming Service*)

Nimeämispalvelu mahdollistaa nimen liittämisen olioon, jolloin kullakin oliolla voi olla yksilöivä ja pysyvä nimi.

Tapahtumapalvelu (*Event Service*)

Tapahtumapalvelu mahdollistaa erilaisten tapahtumatietojen välittämisen olioiden välillä. Mikä tahansa olio voi toimia joko tapahtuman tuottajana (supplier) tai käyttäjänä (consumer). Tapahtumat voidaan välittää joko työntö- (push) tai pyyntö-metodeilla (pull). Palvelu tarjoaa tapahtumakanavan, joka toimii sekä tapahtumien tuottajana että käyttäjänä ja välittää näin saamansa tapahtumat kiinnostuneille olioille.

Elinkaaripalvelu (*Life Cycle Service*)

Elinkaaripalvelu määrittelee palvelut ja käytännöt, joilla luodaan, tuhotaan, kopioidaan ja siirretään olioita.

Olioiden tallennuspalvelu (*Persistent Object Service*)

Tallennuspalvelu tarjoaa yhteiset rajapinnat olioiden varastoimiseen ja niiden pysyvän tilan hallintaan. Tallennuspalvelu toimii usein yhteistyössä muiden palveluiden kanssa, varsinkin nimeämis- ja elinkaaripalvelun.

Transaktiopalvelu (*Transaction Service*)	Transaktiopalvelu tarjoaa rajapinnat, joiden avulla hajautetuissakin järjestelmissä mahdollistetaan atomisten transaktioiden käyttäminen. Nämä rajapinnat huolehtivat siitä, että oliot joko suorittavat kaikki transaktioon liittyvät operaatiot yhdessä tai yhtäkään niistä ei suoriteta.
Samanaikaisuudenhallintapalvelu (*Concurrency Control Service*)	Samanaikaisuudenhallintapalvelu toimii välittävänä tahona, joka huolehtii siitä, että kutsuttavan olion sisäiset tiedot eivät käy ristiriitaisiksi useista käyttäjistä huolimatta. Tämä tehdään rajapinnoilla, jotka mahdollistavat erilaisten lukkojen käyttämisen.
Oliosuhdepalvelu (*Relationship Service*)	Oliosuhdepalvelun avulla voidaan esittää eksplisiittisesti olioiden suhteita toisiinsa. Tämä palvelu tarjoaa kahdenlaisia olioita: suhteita ja rooleja. Oliosuhdepalvelun avulla voidaan navigoida toisiinsa liittyvien olioiden verkossa tarvitsematta aktivoida jokaista läpikäytävää oliota.
Ulkoistamispalvelu (*Externalization Service*)	Ulkoistamispalvelun avulla olio voidaan muuttaa binääridataksi ja tallentaa muistiin, tiedostoon tai siirtää verkon yli. Tämä data voidaan myös muuttaa takaisin olioksi.
Kyselypalvelu (*Query Service*)	Kyselypalvelun avulla voidaan tehdä kyselyitä oliokokoelmista. Kyselyissä käytetään SQL:n tai muiden kyselykielten oliovastineita. Kyselyiden avulla voidaan valita kokoelmasta tietty osajoukko käsiteltäväksi.

Lisenssipalvelu (*License Service*)

Lisenssipalvelun avulla voidaan huolehtia siitä, että jonkin ulkoisen palvelun tuottaja voi luovuttaa tuotteensa toiselle osapuolelle ja varmistua siitä, että saa kohtuullisen korvauksen tuotteensa käytöstä. Lisenssipalvelu mahdollistaa useiden erilaisten lisensiointimallien käyttämisen laskuttamisen apuna.

Ominaisuuspalvelu (*Property Service*)

Ominaisuuspalvelu mahdollistaa erilaisten ominaisuuksien liittämisen olioihin olioiden tyypistä riippumatta. Esimerkiksi dokumenttien käyttäjä voi haluta määritellä dokumentteja tärkeiksi tai vähemmän tärkeiksi. Tällöin tärkeys ei ole dokumenttiolion tyypin mukainen ominaisuus, mutta käyttäjä voi liittää tämän ominaisuuden dokumenttiin. Myös erilaisia laskureita olioiden käytöstä voidaan toteuttaa ominaisuuspalvelulla.

Aikapalvelu (*Time Service*)

Aikapalvelu tarjoaa käyttäjälle mahdollisuuden saada selville kulloinenkin kellonaika ja arvio tämän ajan tarkkuudesta. Lisäksi palvelun avulla voidaan varmistua erilaisten tapahtumien järjestyksestä, luoda aikaan sidottuja tapahtumia ja laskea kahden tapahtuman välinen aika.

Turvapalvelu (*Security Service*)

Turvapalvelu huolehtii siitä, että järjestelmän sisältämä tieto luovutetaan ainoastaan siihen oikeutetuille tahoille. Lisäksi varmistetaan, että tieto päätyy näille henkilöille oikeassa muodossa ja että asiaankuulumattomat eivät voi muuttaa tietoja. Palvelun avulla on kyettävä myös todistamaan kiistattomasti, että käyttäjä on tehnyt jonkin operaation. Edelleen järjestelmän käyttöön oikeutettujen tahojen pääsyä järjestelmään ei saa voida estää tihutöillä.

Kauppapalvelu (*Object Trader Service*)

Kauppapalvelun käyttäminen on tapa löytää olioita hajautetussa järjestelmässä. Oliot voivat mainostaa itseään kauppapalvelussa ja toisaalta niiden käytöstä kiinnostuneet sovellukset voivat kysellä niitä. Yksi tapa löytää olioita Internetistä olisikin perustaa yksi jatkuvasti käytössä oleva kauppapalveluolio, jonka merkkijonoksi muutettua viitettä jaettaisiin vapaasti.

Kokoelmapalvelu (*Object Collections Service*)

Kokoelmapalvelu tukee olioiden keräämistä ryhmiin ja operaatioita kokonaisille ryhmille. Kokoelmat voidaan järjestää ja näin kokoelmasta voidaan valita yksittäisiä olioita indekseillä. Myös erilaisilla hakuavaimilla valinta voi olla mahdollista.

3.6. Asiakassovellus

Asiakassovellus on sovellus, jolla on hallussaan viittaus Corba-olioon. Asiakaskoneessa (tai -prosessissa) on käytettävissä olion tynkä, joka osaa kutsua ORBin avulla palvelinkoneella sijaitsevaa toteutusta. Asiakassovelluksen kannalta olion käyttö muistuttaa mahdollisimman paljon normaalia olioperustaista ohjelmointia.

ORB tarjoaa myös joukon toisenlaisia funktioita, joiden avulla voidaan kutsua sellaisia olioita, joita ei ole määritelty käännösaikana. Tällöin asiakassovelluksella on oltava tietoa käytettävän olion luonteesta ja metodeista. Näiden tietojen perusteella voidaan löytää viittaus tarvittavaan olioon ja sen metodikutsuihin ja näitä voidaan käyttää omien erityisten funktioidensa avulla.

Tavallisesti asiakassovellukset saavat olioviittaukset käytettäväksi metodikutsujen paluuarvoina. Joskus asiakassovellus voi myös itse tarjota oliopalveluita, jolloin se voi antaa viittauksia niihin kutsumiensa metodien parametreina.

3.7. Oliototeutus

Oliototeutus on se osa oliota, joka toteuttaa varsinaisen toiminnallisuuden ja ylläpitää olion tilaa. Olion metodien lisäksi toteutus määrittelee usein myös tavat, joilla olio käynnistetään ja sammutetaan. Toteutus voi käyttää toisia olioita tai muita palveluita tilansa tallentamiseen ja kontrolloimaan oikeuksia olion käyttöön.

Oliototeutus kommunikoi ORBin kanssa selvitettäessä mitä luokan oliota milloinkin kutsutaan, luotaessa uusia olioita ja käytettäessä ORBiin liittyviä palveluita.

Kun olion metodia kutsutaan, ORBin ydin, oliosovitin ja olion runko huolehtivat siitä, että kutsu toimitetaan oikean olion oikealle metodille. Rungon määritelmän mukaisesti toteutusmetodille välitetään myös joitakin parametritietoja. Kun metodi on suoritettu, palauttaa se mahdollisesti jonkin paluuarvon tai poikkeuksen, joka välitetään asiakkaalle.

Yleensä ORBille ilmoitetaan, kun uusi olio on luotu, jotta se tietää, mistä tämän olion toteutus löytyy. Tyypillisesti toteutus myös rekisteröityy jonkin rajapinnan toteuttajaksi, ja se määrittelee käynnistysmenettelyn ellei olio ole jo käynnissä.

3.8. Yhteenveto

Corba on alusta asti suunniteltu mahdollisimman yleiskäyttöiseksi välittäjäohjelmistoksi, jonka avulla asiakas- ja palvelinsovellukset voivat kommunikoida keskenään. Ryhmäohjelmien toteuttamisessa yhtenä olennaisena osana on juuri tämän kommunikointimenetelmän toteuttaminen ja siksi kannattaisikin harkita jonkin valmiin ratkaisun käyttämistä.

Valmiin kommunikointitavan lisäksi Corba voi tarjota joukon palveluita, joiden avulla voidaan helpottaa järjestelmän kehittämistä joko käyttämällä palveluita kehitysvaiheessa väliaikaisena ratkaisuna tai mahdollisesti jopa ottamalla palvelut käyttöön myös valmiissa tuotteessa. Näiden palvelujen yleisesti määritellyn rajapinnan allahan voi olla vaikka minkälainen toteutus, jota voidaan vaihtaa tarpeen vaatiessa. Tämä on mielestäni yksi huomionarvoisimpia seikkoja, joka puolustaa Corban käyttöä.

Kalenteripalvelun kannalta käyttökelpoisia voisivat olla lähes kaikki edellä esitellyt palvelut. Nimeämispalvelun avulla voidaan yhdistää yksilöivät nimet eri kalentereihin ja kalenterimerkintöihin. Näin voitaisiin hakea kalentereita esimerkiksi URL:eja käyttäen.

Tallennuspalvelun avulla voitaisiin tallentaa eri kalenterien ja merkintöjen tilat. Transaktiot ja päällekkäisyydenhallintapalvelut taas ovat välttämättömiä monen käyttäjän ympäristössä, jotta eri olioihin tehtävät muutokset eivät laita koko kalenterijärjestelmää aivan sekaisin.

Oliosuhdepalvelu taas voisi liittää useampia ilmentymiä yhteen tapahtumaan: esimerkiksi koulutustilaisuus joka on määrätty tapahtuvaksi kolmena peräkkäisenä päivänä voitaisiin määritellä yhdeksi tapahtumaksi, joka muodostuisi näistä kolmelle eri päivälle sijoittuvista varauksista. Ulkoistamispalvelun avulla voitaisiin siirtää kalenterivarauksia vaikka PDA-laitteeseen. Kyselypalvelu helpottaisi kalenterin tilan tutkailua esimerkiksi vapaata aikaa etsittäessä.

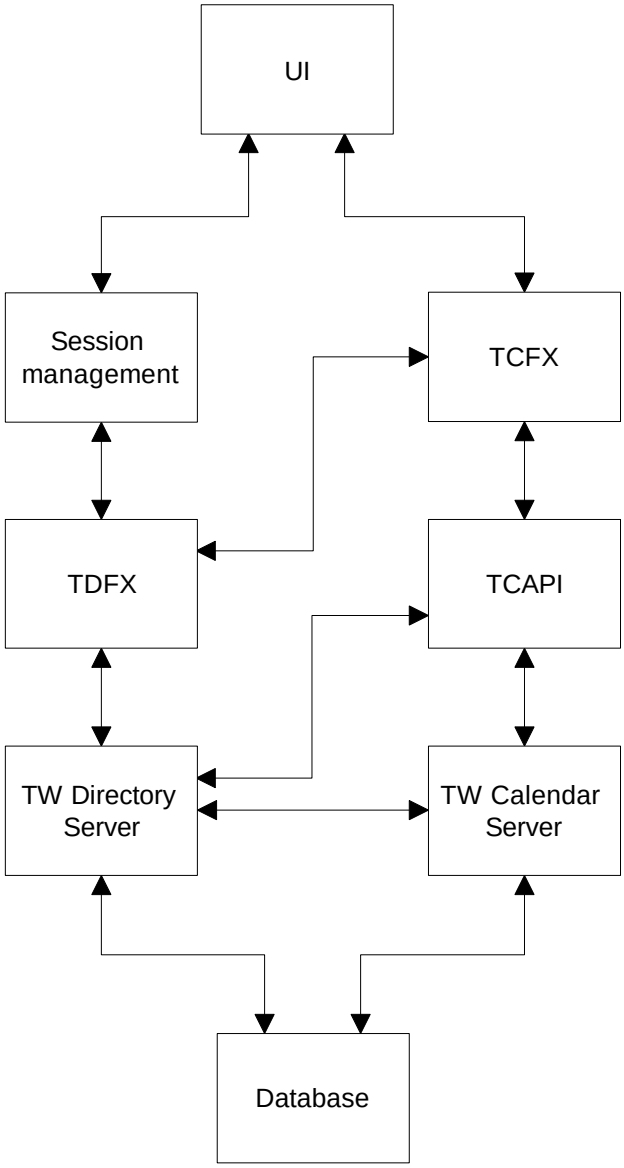
Lisenssipalvelu olisi tuotteen myyjää helpottava ominaisuus, aikapalvelun avulla taas voitaisiin huolehtia siitä, että koko järjestelmällä on yhteinen aika.

Palveluiden lisäksi Corbaa puolustava ominaisuus on olioperustaisuus: nykyisin käyttöliittymiä toteutetaan oliopohjaisina, tietorakenteet ovat oliopohjaisia ja ohjelmointikielet ovat oliopohjaisia. Tästä syystä voitaisiin kuvitella, että tiedon esittämisessä oliopohjaisuus on hyödyllistä.

4. Case: TeamWARE

4.1. Nykyinen arkkitehtuuri

Alla (Kuva 3) on esitetty kalenteripalvelut sisältävä TeamWARE-järjestelmä yksinkertaisimmillaan.



Kuva 3 : TeamWARE Office-järjestelmän arkkitehtuuri

Kaiken pohjana on TeamWAREn kehittämä DataCom-objektitietokanta. Normaalien relaatiotietokantaan talletettavien tietojen lisäksi se osaa hallita myös tiedostoja.

Tietokannan kanssa sellaisenaan keskustelevat palvelinohjelmistot, joita ovat mm. hakemisto- ja sovelluspalvelimet. Hakemisto on eräänlainen ryhmäosoitekirja, johon on talletettu tiedot kaikista järjestelmän käyttäjistä. Tämän lisäksi käyttäjät itse voivat lisäillä ns. ulkopuolisia käyttäjiä, joille voidaan esim. lähettää sähköpostia järjestelmästä.

Sovelluspalvelimet osaavat noutaa tietokannasta sovelluskohtaista tietoa, kuten kalenterivarauksia tai sähköposti- tai forum-viestejä. Sovelluspalvelinten rajapinta on määritelty TeamWAREn luomalla ASN.1:n kaltaisella TASN-määrittelykielellä.

Useimmissa TeamWARE-sovelluksissa asiakassovellukset keskustelevat palvelinten kanssa sovelluskohtaisten T*FX-modulien kautta, kuten Tiimi-postin (TeamWARE Mail) TMFX, Tiimi-foorumin (TeamWARE Forum) TFFX tai Tiimi-arkiston (TeamWARE Library) TLFX. Nämä moduulit noutavat raakaa dataa tietokannasta ja muuttavat tämän tiedon olioiksi, joita sitten sovellukset käyttävät.

Kalenteri on tässä suhteessa poikkeuksellinen sovellus; sitä varten on tehty oma kalenteri-API, TCAPI, joka toteuttaa C-rajapinnan tietokantaan. TCAPI osaa mm. hallita useampien palvelinten järjestelmää sekä käsitellä tietokannasta saatavan tiedon jo kohtuullisen hyvin sovellusten hallittavaan muotoon. Kalenterin TCFX pääseekin helpolla, sillä sen ei tarvitse tehdä muuta kuin valmista C-rajapintaa käyttäen pyytää palvelimilta tietoa ja muuttaa se olioiksi.

Edellinen kuva siis on kalenterijärjestelmä yksinkertaisimmillaan. Systeemi muuttuu huomattavasti monimutkaisemmaksi, kun otetaan mukaan TeamWAREn muut sovellukset, Posti, Arkisto, Foorumi sekä agentit. Kaikkien sovellusten pitäisi olla tietoisia toisistaan ja usein ne jopa haluavat lähetellä toisilleen viestejä, joista yleisimpänä on sähköpostiviestin lähettäminen mistä tahansa muusta sovelluksesta. Lisäksi agenttien pitää päästä kä-

siksi kaikkien sovellusten tietokantoihin. Yksinkertainen ongelma ei ole myöskään se, miten 32-bittisessä suojatun muistin käyttöjärjestelmässä hallitaan käyttöoikeudet, kun ei haluta, että jokaisen sovelluksen käynnistämisen yhteydessä täytyy erikseen kirjoittautua sisään järjestelmään.

4.2. Suunnittelunäkökohtia

Nykyinen järjestelmä on toimiva mutta vanhanaikainen, mikä näkyy esimerkiksi siten, että tietokannan sisältämä data muutetaan olioiksi vasta asiakaskoneessa. Ohjelmoijan kannalta elegantimpi ratkaisu olisi luoda jo palvelimien päälle oliorajapinta, jota asiakasohjelmat ja agentit voisivat käyttää suoraan. Yhtenä lähtökohtana siis on selvästikin se, että halutaan luoda oliorajapinta, jota kaikki järjestelmän osat voivat käyttää.

Toisena lähtökohtana on tarve standardeille. OMG on esittänyt ns. esityspyynnön, RFP:n (*Request for Proposal*) [CORBA, 97], eli se pyytää jäsenorganisaatioita tekemään esityksiä kalenteripalvelun määritelmäksi. Tähän pyyntöön voi periaatteessa suhtautua ainakin kolmella tavalla:

1. Sen voi kokonaan jättää huomiotta.

   Tässä lähestymistavassa hyvinä puolina on joustavuus ja nopeus. Kun ei tarvitse odottaa virallisen määritelmän valmistumista, voi määrittelyn tehdä niin nopeasti kuin ehtii ja heti alkaa toteuttaa sitä.

   Huonona puolena on se, että näin rakennettava järjestelmä ei kiinnosta ketään muuta ellei siinä ole jotain todella poikkeuksellista tarjottavaa. Lopputuloksena voi siis olla omaan käyttöön jäävä järjestelmä ja vastarannan kiisken maine.

   Omassa työssäni olen valinnut tämän lähestymistavan.

2. Voi odottaa määrittelyn valmistumista ja toteuttaa sitten valmiiksi määritellyn rajapinnan.

   Näin joutuu odottamaan määrittelyn valmistumista, siihen ei voi vaikuttaa mitenkään ja lopputulos voi olla jotain aivan muuta, kuin mitä omat tähänastiset toteutukset ovat olleet. Näin saattaa joutua tekemään

paljonkin ylimääräistä työtä omien systeemien saattamiseksi sellaiseen kuntoon, että määritelty rajapinta on toteutettavissa.

Toisaalta taas näin säästää vaivaa ja rahaa, joita määrittelyn tuottamiseen muuten kuluisi.

3. Voi osallistua määrittelyprosessiin ja näin vaikuttaa lopputulokseen. Vaikuttamisen mahdollisuudet eivät ole välttämättä kovin suuret, sillä Corba-standardin luomisessa täytyy kuitenkin tukeutua jo olemassa oleviin standardeihin ja ehdotuksiin, kuten vCalendar ja iCalendar. Toimittajakohtaisiin toteutuksiin vetoaminen ei myöskään ole viisasta. Joka tapauksessa määrittelyyn täytyy uhrata resursseja, mikä on aina riski, varsinkin kun varsinaisesta hyödystä ei ole mitään takeita.

   Hyväksi puoleksi voidaan laskea se, että mahdollisuus vaikuttaa on kuitenkin parempi kuin ei mahdollisuutta. Lisäksi hyvän esityksen tekeminen parantaa yrityksen imagoa.

   Tämän vaihtoehdon osalta päätöksen tekeminen lienee jo myöhäistä, sillä lopullinen RFP (Request for Proposal) julkaistiin joulukuun 5. päivänä 1997. Alkuperäisen aikataulun mukaan ensimmäiset esitykset piti jättää maaliskuussa 1998 ja lopullisen rajapinnan piti ilmestyä marraskuussa 1998. Valitettavasti kaikki toiminta tämän aiheen ympärillä näyttää pysähtyneen, sillä OMG:n aihetta koskevalla sivulla ei ole aikoihin tapahtunut yhtään mitään.

4.3. TeamWARE Calendar Corballa

4.3.1. Oliomallin suunnittelu

Suurten ohjelmistoprojektien toteuttamisen perusteena on aina huolellinen suunnittelu; erityisen tärkeätä suunnittelu on hajautetuissa järjestelmissä, joissa kaikki palvelinrajapintoihin tehtävät muutokset heijastuvat moninkertaisina asiakassovelluksiin. Corbaa käytettäessä on luonnollisesti mahdollista käyttää dynaamista rajapinnan määritystä, mutta se on tehotonta ja työlästä ohjelmoida. Rajapintojen huolellinen suunnittelu on siis ensiarvoisen

tärkeää mutta toisaalta helpottavaa, koska samaa oliomallia käytetään sekä asiakas- että palvelinohjelmassa.

Jotta oliomalli muodostuisi jo suunnittelun aikana mahdollisimman täydelliseksi, kannattaa suunnittelussa hyödyntää skenaarioita, jotka kuvaavat erilaisia käyttötilanteita olioiden toisilleen lähettämien viestien avulla.

Suunnittelussa käytin työvälineenäni Rational Softwaren ohjelmaa Rational Rose, jonka ominaisuuksiin kuuluu mm. juuri skenaariokaavioiden avulla suunnittelu sekä OMT-, UML- ja Booch-notaatiot. Omassa suunnittelussani käytin OMT-notaatiota.

Kalenterin oliomallia suunnitellessani päätin aloittaa yhden käyttäjän kalenterin tutkimisesta. Tästä laajensin mallia käsittämään useita kalentereita ja viimeisessä vaiheessa otin käyttöön vielä järjestelmän käyttöoikeuksien hallinnan ja valvonnan. Seuraavassa on esitelty nämä kolme suunnitteluvaihetta.

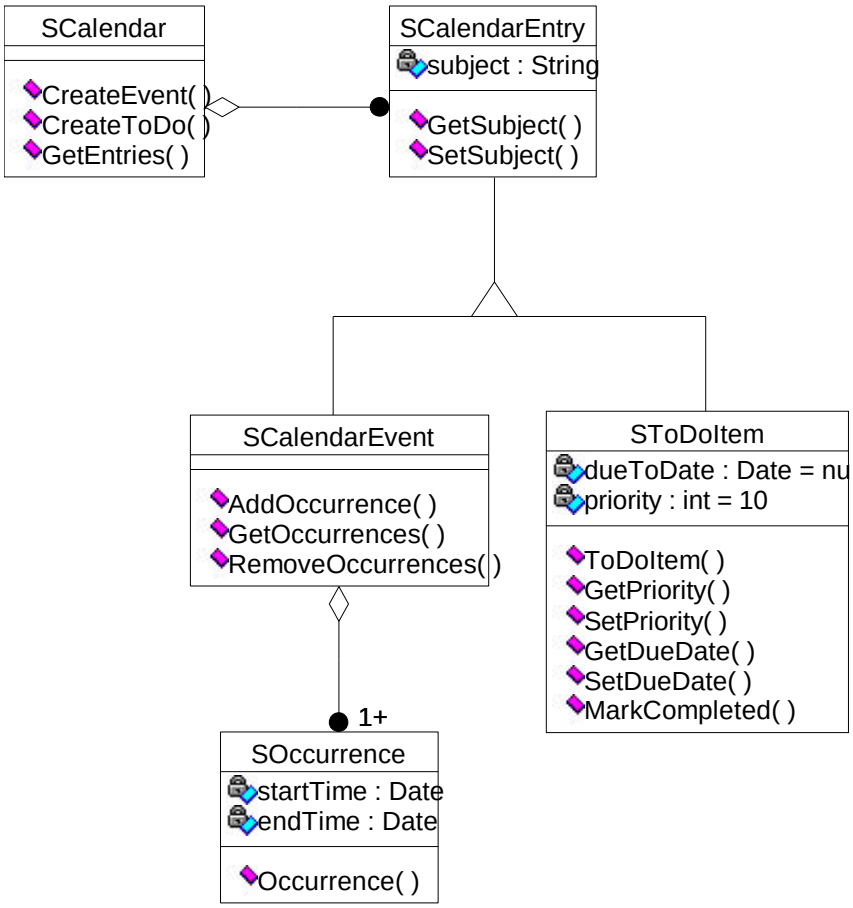
4.3.2. Yhden käyttäjän kalenteri

Yhden käyttäjän kalenteri on esitettävissä hyvin yksinkertaisella mallilla. Tarvittavia luokkia ovat Calendar, joita yhtä sovellusta kohti ei tarvita kuin yksi. Kalenteriin voidaan tehdä merkintöjä, joita on kahta laatua: tapahtumia (Event) ja tehtäviä (ToDoItem).

Tapahtuma voi esiintyä kalenterissa useampia kertoja, näitä kutsutaan esiintymiksi (Occurrence). Esiintymiä on voitava muokata yksitellen ja ne on voitava poistaa tietyltä aikaväliltä tai jopa kokonaan.

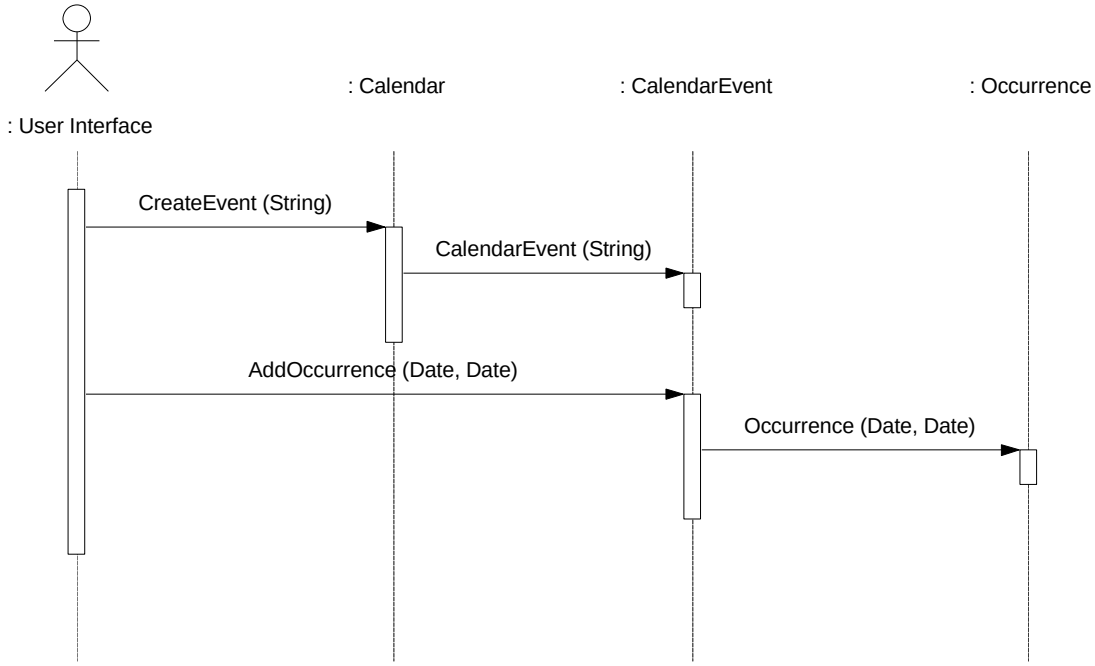
Tehtäville on voitava asettaa jonkinlainen tärkeysjärjestys ja päivämäärä, jolloin tehtävän täytyy viimeistään olla suoritettu.

Seuraavassa kuvassa (Kuva 4) on esitetty yksinkertainen kalenterisovelluksen oliomalli.



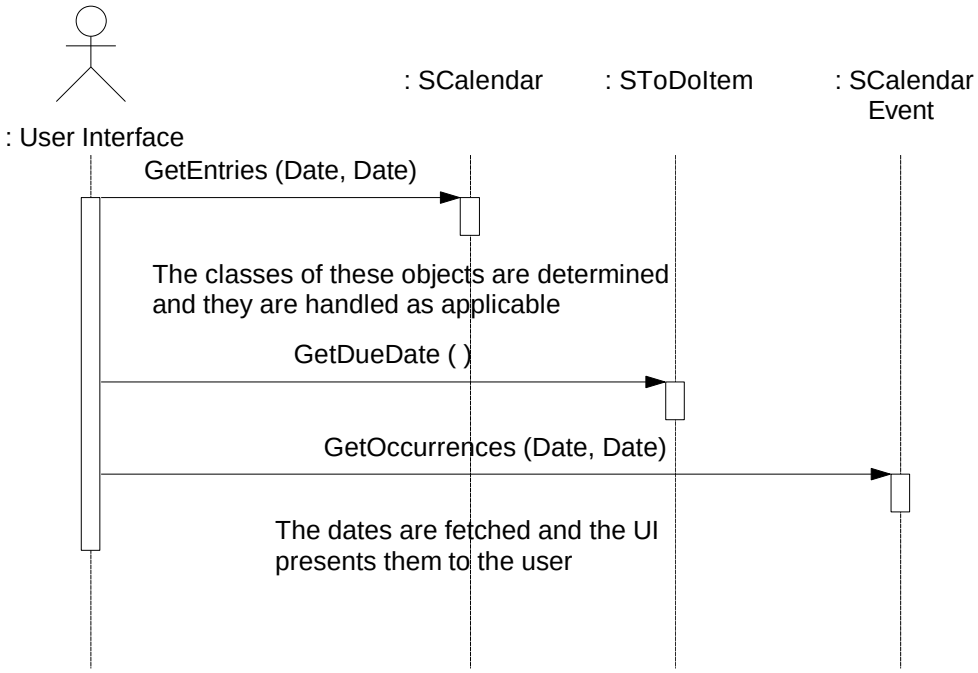
Kuva 4 : Yhden käyttäjän kalenterin oliomalli

Skenaariokaavioissa esitetään tietty tapahtuma olioiden kannalta. Kuva 5 esittää yksinkertaisen skenaarion, jossa käyttäjä luo kalenteriinsa tapahtuman ja tähän tapahtumaan liittyvän ajankohdan. Kuva 6 esittää tapahtumasarjaa, jossa sovellus kysyy kalenterilta tietyn ajanjakson merkinnät voidakseen esimerkiksi esittää ne käyttöliittymässään.



Kuva 5 : Tapahtuman luominen kalenteriin

Kun tapahtumaa aletaan luoda kalenteriin, luodaan ensin tapahtuma-olio, johon sitten lisätään tarvittavat esiintymät.



Kuva 6 : Tietojen kysyminen kalenterilta

Sovellus pyytää kalenterilta merkinnät tietyllä aikavälillä. Nämä merkinnät kalenteri antaa *Entry*-olioina, joista sovellus määrittelee, onko kyseessä *ToDoItem*- vai *CalendarEvent*-luokan edustaja. Näiltä olioilta sovellus kysyy tarkemmat ajankohdat, joiden perusteella ne voidaan esittää käyttöliittymällä.

4.3.3. Monen käyttäjän kalenteriympäristö

Kun siirrytään yhden käyttäjän kalenterista monen käyttäjän kalenteriympäristöön, täytyy ottaa huomioon joitakin uusia asioita. Esimerkiksi ryhmäkalenterisovellusten luonteeseen kuuluu, että kuka tahansa käyttäjä voi katsella muidenkin käyttäjien kalentereita sekä tehdä niihin omia varauksiaan. Toisaalta taas käyttäjät voivat vastata näihin muiden tekemiin varauksiin (tai oikeammin varauspyyntöihin) myöntävästi tai kieltävästi.

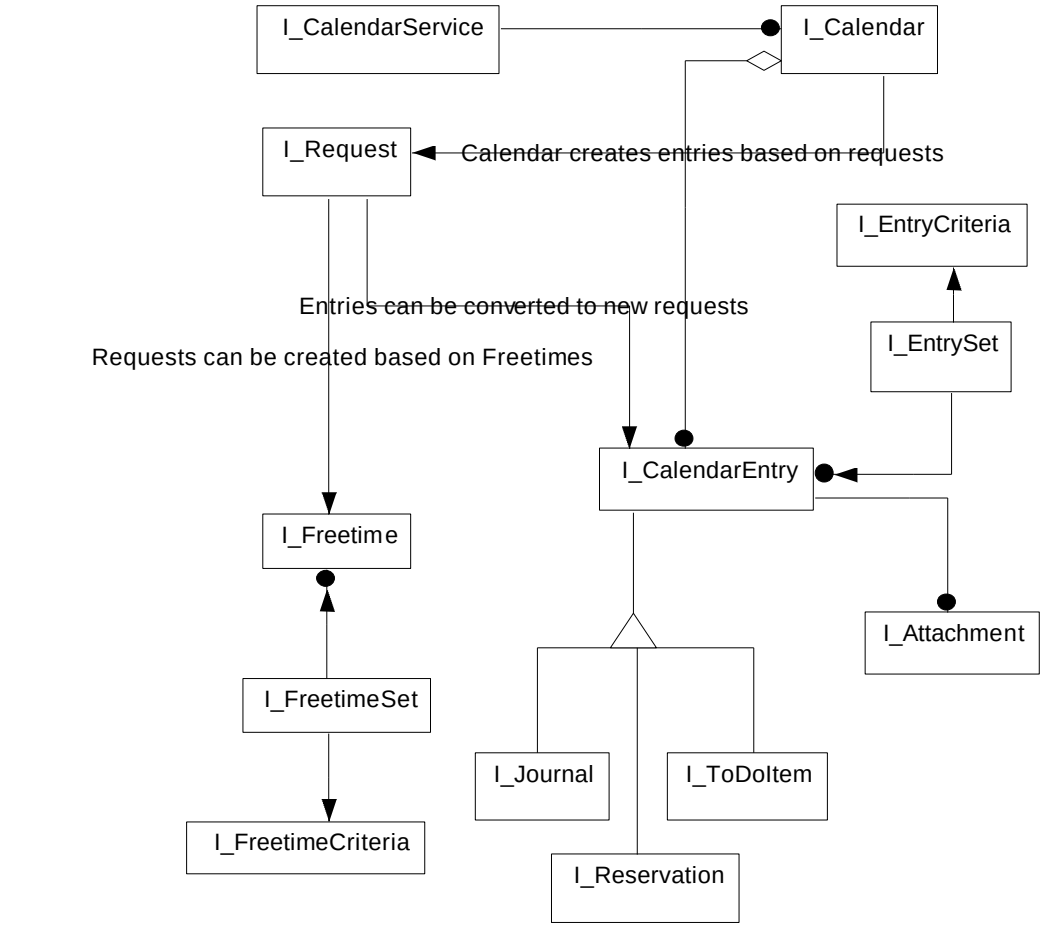
Koska kalenterimerkintä voi olla useammassakin kuin yhdessä kalenterissa, ei näitä merkintöjä voi suoranaisesti enää määritellä yhden kalenterin osiksi vaan on kannattavampaa luoda jonkinlainen varasto, jossa merkinnät säilytetään. Näin merkintöjä ei välttämättä enää luoda kalenterin vaan varaston kautta. Toisaalta täytyy olla olemassa mekanismi, jolla tapahtumien osanottajia voidaan lisätä ja poistaa. Täytyy ottaa myös huomioon, että kalenterinkäyttäjän on voitava päättää jokaisen esiintymän kohdalla erikseen, haluaako hän osallistua vai jättää osallistumatta.

Monen käyttäjän ympäristössä luonnollisesti tarvitaan myös jonkinlaista hakemistoa, josta voidaan kysyä olemassa olevien käyttäjien tietoja.

Eräs huomioonotettava seikka tulee myös esiin usean käyttäjän ympäristössä: suorituskyky. Usein oliomallin eleganttiudesta joudutaan tinkimään suorituskyvyn pitämiseksi hyvällä tai edes siedettävällä tasolla. Näin esimerkiksi kalenterimerkintöjen hakeminen kalenteriolion kautta ei ole tehokasta, jos esimerkiksi haku tehdään useasta kalenterista ja sama tapahtuma löytyy näistä kaikista. Tällöin sama tapahtuma haetaan useaan kertaan ja

useiden ilmentymien käsittely täytyisi toteuttaa asiakassovelluksessa. Tehokkaampaa on tehdä kyselyt jälleen varaston kautta käyttäen jonkinlaisia kriteeriolioita parametreina.

Jotkin Corba-toteutukset mahdollistavat myös asynkroniset metodikutsut, jolloin käyttäjän ei tarvitse jäädä odottamaan pitkäkestoisen operaation suoritusta. Koska emme kuitenkaan halua sitoa toteutusta mihinkään yksittäiseen Corbaan, kannattaa jo rajapintaan määritellä tämän kaltaiset taustaoperaatiot. Näiden osalta lopputulos muistuttaa hiukan Java 1.1:n tapahtumankäsittelymekanismia, vaikkakin tapahtumien vastaanottajien käyttö onkin enemmän *ad hoc*-luontoista.



Kuva 7 : Monen käyttäjän kalenterin oliomalli

Kuva 7 esittää monen käyttäjän kalenteripalvelua. Heti sovelluksen käynnistyessä on luotava *I_CalendarService*-tyyppinen kalenteripalveluolio.

Tämän kautta voidaan luoda tai tuhota kalentereita ja niitä voidaan hakea. Kalenteripalveluun voidaan kiinnittää kuuntelijoita, jotka reagoivat kalenteritietokannassa tapahtuneisiin muutoksiin tai muutokseen yhteydessä (yhteyden luonti tai katkeaminen).

Monen käyttäjän kalenterimaailmassa kuka tahansa ei saa tehdä varauksia suoraan mihin tahansa kalenteriin, vaan käyttöön otetaan termi 'varauspyyntö'. Esimerkiksi käyttäjä A:n halutessa varata aikaa tapaamiseen omaansa ja käyttäjän B kalentereihin, hän lähettää B:lle varauspyynnön. Tämän pyynnön B voi sitten joko hyväksyä tai hylätä. Hyväksymistapauksessa varauspyynnön mukainen varaus sijoitetaan kalenteriin ja tästä lähetetään palaute A:lle. Omaan kalenteriin tehdyt varaukset hyväksytään aina automaattisesti.

Mikäli jonkin kalenterin omistaja haluaa myöhemmin tehdä muutoksen aiempaan varaukseensa, hän voi kutsua I_CalendarEntryn metodia convertToRequest, joka palauttaa I_Request-objektin, joka voidaan taas antaa kalenterioliolle varauspyyntönä. Näin tehdyt pyynnöt korvaavat aiemman varauksen joko suoraan tai pyyhkivät vanhan varauksen pois ja jäävät odottamaan uutta hyväksyntää kalenterin omistajalta.

Monen käyttäjän ympäristössä on hyödyllistä olla käytettävissä jonkinlainen mekanismi, jolla voidaan etsiä vapaata aikaa useista kalentereista yhtä aikaa ja tehdä varaus tälle löydetylle vapaalle ajalle. Tätä varten mallissa on I_FreetimeSet-olio, jonka avulla näitä hakuja voidaan tehdä.

Vapaata aikaa haettaessa luodaan FreetimeCriteria, joka täytetään tarpeellisilla tiedoilla kuten haettavan ajan pituus, aikajakso, jolta aikaa haetaan ja halutut kalenterit. Haussa voidaan myös määritellä ryhmiä, joissa vapaata aikaa haetaan vain yhdestä kalenterista. Esimerkiksi kokousta järjestettäessä vaihtoehtona voi olla kolme neuvotteluhuonetta, joista vain yhteen tehdään varaus.

Tyypillisesti asiakas-palvelin-ympäristöissä kuten TeamWARE-kalenterissa tällaisen haun tekeminen on kohtuullisen nopea operaatio. Toisaalta taas yleisen kalenterimallin suunnittelussa täytyy ottaa huomioon

myös Internet-standardeihin perustuvat vaihtoehdot kuten iTip ja iCalendar. Näissä varauspyynnöt ja varaustilannekyselyt tehdään sähköpostitse ja niihin vastauksen saaminen voi kestää kauankin. Tällöin ei ole järkeä jättää käyttäjää odottamaan vapaiden aikojen haun päättymistä, vaan hakupyyntö tehdään kalenteripalvelulle ja samassa yhteydessä annetaan parametrinä myös tätä tilannetta varten nimetty kuuntelija, jolle ilmoitetaan, koska haku on suoritettu. Haun kestäessä käyttäjä voi käyttää kalenterisovellustaan mihin tahansa muuhun ja hän saa tiedon haun päättymisestä käyttöliittymälleen.

Vapaan ajan haun päätyttyä käyttäjä saa käsiinsä FreetimeSetin, joka sisältää tiedot käytettävissä olevista vapaista ajoista sekä siitä, mitkä kalenterit näinä aikoina ovat vapaina. Vapaa-aikatieto voidaan muuttaa varauspyyntöolioksi, joka puolestaan voidaan lähettää asianosaisille kalentereille.

Edellä kuvatuista tehokkuussyistä johtuen malli ei juurikaan muistuta alkuperäistä yhden käyttäjän kalenterimallia.

4.3.4. Kalenteriympäristö käyttöoikeuksilla

Todellisessa maailmassa käyttäjällä täytyy olla mahdollisuus rajoittaa toisten käyttäjien oikeuksia nähdä kalenterinsa tila ja tehdä siihen varauksia. Luonnollisesti kaikilla ei ole edes pääsyoikeuksia koko järjestelmään.

Yksi Corban palveluista on Security Service, jonka huoleksi voidaan jättää käyttäjän autentisointi ja joissakin toteutuksissa jopa yksittäiseen olioon liittyvät oikeudet. Tämä palvelu ei kuitenkaan kuulu Corban peruskomponentteihin, joten senkään olemassa oloon ei voida luottaa. Siksi nämä ominaisuudet täytyy ottaa huomioon jo oliomallissa.

Lähdin ratkaisemaan ongelmaa siten, että periytin edellisen oliomallin kalenteriluokasta uuden luokan, *I_AuthenticatedCalendarin*, joka sisältää käyttäjän oikeudet järjestelmään. Tätä perittyä kalenteria käytetään aina eräänlaisena kahvana, jolla eri palveluihin päästään käsiksi. Lisäksi järjestelmästä uloskirjautumiseen käytettävä metodi löytyy ainoastaan tästä autentisoidusta kalenterioliosta.

AuthenticatedCalendar omaa muun muassa tiedot siitä, minkälaisia oikeuksia kyseisellä kalenterilla on järjestelmään ja lisäksi kahden kalenteriolion välillä voi olla määriteltyjä oikeustasoja, jotka määräävät esimerkiksi sen, voiko yksi kalenteri nähdä toisen kalenterin merkintöjä ja näiden otsikkotietoja tai voiko kalenterin kautta suoraan muokata toisen kalenterin sisältöä.

4.4. Toteutus

4.4.1. Työkalut ja ympäristöt

Edellä kuvatussa kalenteriympäristössä kalenteripalveluoliot olisi kannattanut toteuttaa suoraan TCAPI-rajapinnan päälle. Erinäisistä syistä TeamWARE toteutti kuitenkin TCAPIn päälle tarkkaan määritellyn kalenterikutsurajapinnan, joka sisälsi kalenteripalveluiden lisäksi myös käyttäjän autentisoinnin ja yleisimmät hakemistopalvelut, kuten kalenterien haun tietokannasta. Tämän rajapinnan käyttö olioiden toteuttamiseen tuntui siis luonnollisimmalta vaihtoehdolta.

Palvelin toteutettiin käyttäen Microsoftin Visual C++:aa ja ICL:n Dais-Corbaa. Asiakassovellukset taas yritin toteuttaa Daiss/C++-version lisäksi myös Sunin Java Platform 2:lla, mutta en valitettavasti onnistunut tässä kovin hyvin. IDL-koodi generoitiin Rational Rosella suoraan sillä suunnitelluista malleista.

4.4.2. Työ

Kalenterin oliomallia suunnitellessani lähtökohtana oli yksinkertainen monen käyttäjän kalenteriympäristö, johon liitin joitakin piirteitä käyttöoikeuksia sisältävästä järjestelmästä.

Suunnittelin kalenteripalvelun oliomallin alunperin Java-kielistä toteutusta silmälläpitäen. Tästä aiheutui ongelmia, kun aloin kääntää Rosella tuotettuja IDL-kuvauksia C++-kielisiksi koodeiksi. Huomasin siis heti aluksi, että oliomalli kannattaa suunnitella aina mahdollisimman kieliriippumatto-

masti, ja tämä onnistuu parhaiten ohjelmointikielestä riippumatonta IDL:ää käyttäen, sillä käännökset siitä eri ohjelmointikieliin ovat yksikäsitteisiä.

Rose tuotti IDL-koodin, jonka käänsin Daisin IDL-kääntäjällä. Valitettavasti tuotettu IDL ei suostunut sellaisenaan kääntymään, mikä johtui pitkälti C++:n kaltaisesta käännöstavasta. Ristiinriippuvuudet tiedostojen välillä aiheuttivat ongelmia ja jouduin *include*-lausekkeiden lisäksi tekemään useita rajapintojen esittelyjä tiedostoihin, jotta luokat tunnistaisivat toisensa. Tämä taas aiheutti suunnattoman joukon varoituksia ja johti rumaan koodiin.

Kun sain ensimmäisen kerran kaikki IDL-määritelmät kääntymään C++ -koodiksi ja yritin käännöstä Visual C++:lla, sain tulokseksi joukon virheilmoituksia, joiden luonteesta johtuen epäilin syynä olevan väärän kääntäjäversion. Tällöin poistin koneeltani version 4.1 ja asensin version 4.0, mutta ongelma ei korjautunut.

Tässä vaiheessa päätin kopioida erillisissä tiedostoissa sijainneet rajapintamääritelmät yhteen suureen tiedostoon ja sijoitin ne yhden IDL-modulin sisään. Tämän ratkaisun seurauksena useat virheilmoitukset poistuivat. Jäljelle jäi virheilmoitus, jonka mukaan *typedef*-komennolla ei voi määritellä muuttujatyyppiä nimeltä *Request_ptr*. Tämän virheilmoituksen syyksi paljastui lopulta Corban oma samanniminen pseudorajapinta. Ongelma korjaantui, kun muutin oman luokkani nimeksi CalendarRequest.

Seuraavaksi poistin luokat koonneen moduulin määritelmien ympäriltä ja kohtasin uuden virheen. Kalenterivarausjoukossa tapahtuneista muutoksista tiedottava tapahtuma *SetEvent* meni päällekkäin Win32-kernelin funktion kanssa. Jouduin siis muuttamaan tämänkin rajapinnan nimeä IDL-koodissa.

Tähänastiset ongelmani olivat siis johtuneet kolmesta syystä:

1. Kehno IDL-generaattori

   Ainakaan kyseinen työkalu ei ole ottanut täysin huomioon IDL-kääntäjien tapaa kääntää koodia; aidosti IDL:n huomioonottava IDL-generaattori osaisi automaattisesti tuottaa koodia, joka sellaisenaan kelpaisi kääntäjälle.

2. Kehno IDL-kääntäjä

   Koska olin työskennellyt pääasiassa Javan kanssa yli vuoden ajan ennen kuin aloin toteuttaa kalenteripalvelua, joutuminen uudelleen vastakkain C++-tyylisen esikääntäjän kanssa aiheutti melkoisen kulttuurishokin. Java ei tunne esikäännöstä ja luokkien, metodien ja muuttujien esittely voidaan suorittaa lähes missä järjestyksessä tahansa.

   Daisin esikääntäjä tarjoaa oman komennon import, joka toimii samaan tapaan kuin Javan import-komento, eli se ei suoraan sisällytä tarvittavaa koodia käännettävään tiedostoon, vaan hakee sieltä rajapintojen määritelmiä vasta, jos esitellyn rajapinnan määritelmää ei löydy muuten. Tämä komento on kuitenkin Dais-riippuvainen, ja koska tarkoituksenani oli käyttää täsmälleen samaa IDL-koodia myös muiden kääntäjien kanssa, en voinut turvautua sen käyttöön. Idea on siis hyvä, mutta tässä käytössä epäkelpo.

   Olisi myös toivottavaa, että IDL-kääntäjät osaisivat varoittaa, jos käytetään joitain Corban peruspalveluille varattuja nimiä. Tosin tämän toteuttaminen on teknisesti hankalaa ja tarve vähenee, kun käyttäjä oppii uutta.

3. Kehno tekijä

   Windows-ympäristössä ohjelmoitaessa kuuluu Win32-APIn yleisien funktioiden tunteminen yleissivistykseen. Lisäksi tämä oli ensimmäinen kerta, kun tein tällaisen virheen, enkä osannut heti yhdistää virheilmoitusta tällaiseen virheeseen koodissani. Tosin kauemmin alalla toiminut kolleganikaan ei osannut auttaa tilanteessa, joten ongelmasta voidaan syyttää myös kehitysympäristöä.

Kun lopulta sain IDL-koodin käännettyä C++-kielisiksi runkoluokiksi, aloin kirjoittaa toteutuksia, jotka tuottavat varsinaisen toiminnallisuuden. Tämä oli varsin suoraviivaista, sillä generoidut rungot sisälsivät abstraktit luokat, joissa kaikki metodit oli ainoastaan ilmoitettu, mutta joita ei ollut määritelty millään lailla. Näistä luokista periytin toteutusluokat. Tässä vaiheessa ei esiintynyt minkäänlaisia ongelmia.

Oliototeutuksia työstäessäni törmäsin vain pariin piirteeseen, joka rikkoi illuusion tavallisesta paikallisten olioiden ohjelmoinnista: merkkijonoja palautettaessa ei voida normaaliin C++-tyyliin vain palauttaa merkkijonon alkuun viittaavaa osoitinta, vaan merkkijono on kopioitava käyttäen Corban omaa *string_dup*-funktiota. Myöskään uusien olioiden viitteitä ei voinut palauttaa sellaisenaan, vaan oliosta oli Corban *narrow*-funktiota käyttäen luotava perusluokan viite.

Nämä piirteet oppii kuitenkin nopeasti ja tämän jälkeen ohjelmointi onkin hyvin yksinkertaista. Sen jälkeen, kun olin saanut ensimmäisen kerran asiakassovelluksen kutsumaan oliototeutusta, kaikki alkoikin sujua huomattavasti jouheammin enkä joutunut jatkossa kiinnittämään lainkaan huomiota näiden kahden ohjelman väliseen kommunikointiin.

Asiakassovelluksen kehittäminen oli erittäin suoraviivaista. Oikeastaan ainoat tilanteet, joissa jouduin huomaamaan käyttäväni Corbaa, olivat alkuperäisen kalenteripalvelun hakeminen kauppiaspalvelusta ja käytettyjen olioiden vapauttaminen Corban *release*-metodilla.

Kun olin saanut asiakassovelluksen toimimaan, yritin toteuttaa saman Javalla. Java 2 –ajoympäristö sisältää yksinkertaisen ja kevyen ORB:n ja erilli-

senä pakettina Javasoftin sivuilta ladattava *idltojava*-kääntäjä ymmärsi täysin muutoksitta DAISin kanssa käyttämäni IDL-tiedoston.

Java-asiakassovelluksen koodin kirjoitin käyttäen sapluunana C++-asiakassovelluksen koodia, jonka kopioin ja kommentoin Java-lähdekooditiedostoon. Näiden kommenttien alle kirjoitin Java-versiot kustakin metodikutsusta korvaten C++-osoittimien nuolinotaatiot Javan pistenotaatioksi. Tämän jälkeen koodi kääntyi jälleen sellaisenaan.

Ongelmia alkoi ilmentyä, kun yritin siirtää olioviitteen Daisilla tehdystä palvelimesta Javalla tehtyyn asiakkaaseen. Käytin tähän Corban *object_to_string*- ja *string_to_object*-metodeja, mutta jostain syystä Javalla ladattua olioviitettä ei osattu tulkita olioksi. Ensin syynä oli väärin tiedostoon kirjoitettu merkkijono ja kun korjasin tämän virheen, kieltäytyi Java 2:n ORB tunnistamasta olioviitettä olioksi. Lopulta jouduin päättelemään, että joko Daisin tai Java Platformin IIOP-tuessa on puutteita; todennäköisemmin ensimmäisessä, mutta koska käytettävissäni ei ollut muita Corba-tuotteita, en voinut testata päätelmääni.

4.5. Yhteenveto

Ongelmista huolimatta Corba vaikuttaa mielenkiintoiselta ja houkuttelevalta konseptilta ja sen käyttäminen asiakas-palvelin-sovellusten toteuttamiseen on pienistä varsin helppoa. Tosin ainakaan Daisia ei ole integroitu esimerkiksi Visual C++:aan samalla lailla kuin DCOM, mutta toisaalta taas sitä ei ole myöskään rajoitettu tähän yhteen ainoaan kehitysvälineeseen. Lisäksi Daisympäristöt löytyvät sekä 32-bittiseen Windowsiin että Solarikseen, joten pelkästään tällä yhdellä toteutuksella voidaan järjestää keskusteluyhteys näiden kahden ympäristön välille. Valitettavasti ICL on lopettanut Daisin kehityksen.

Koska en saanut tehtyä 'kilpailevia' asiakassovelluksia, ei nopeusvertailuissa tuntunut olevan mitään mieltä. Tosiasiassa yhdellä koneella prosessien välinen kommunikointi oli kuitenkin riittävän nopeaa, 17 operaation – näistä yksi epäonnistunut ja tuotti vastauksen sijaan poikkeuksen – suorit-

taminen vei aikaa pahimmillaankin alle 0,8 sekuntia. Yhden koneen sisäinen kommunikointi ei luonnollisesti kerro tiedon kulun nopeudesta verkossa, mutta toisaalta sekä asiakas että palvelin pyörivät samalla koneella, joten suorituskyky voi jopa parantua kun prosessit siirretään ajettaviksi eri koneilla.

Corba tekee asiakas-palvelin-sovellusten toteuttamisen helpoksi: kommunikoinnin toteuttamiseen ei tarvitse kiinnittää huomiota, sillä kaikki löytyy valmiina. Mikäli IIOP todella tekee sen minkä lupaa – en ole yhden epäonnistuneen kokeilun jälkeen valmis heittämään kirvestä kaivoon – on käytettävissä järjestelmä, jonka avulla eri käyttöjärjestelmien, ohjelmointikielten ja kehitysvälineiden rajat eivät enää olekaan ylittämättömiä.

4.6. Päätelmiä

Suurin syyni pitää ohjelmoinnista on onnistumisen ilo ja Corba-kalenterin ohjelmoiminen tuotti paljonkin onnistumisia. Oli hienoa nähdä ensimmäisen kerran, kuinka tekemältäni palvelimelta lähti tieto, joka pääsi perille tekemääni asiakkaaseen. Ja oli hienoa saada käännettyä Java-asiakas lähes saman näköisenä kuin mitä C++-versio oli ollut – olkoonkin, että asiakas ei koskaan pystynyt kommunikoimaan palvelimen kanssa.

Corban kanssa työskennellessäni huomasin, että se on suorastaan tuskastuttavan oliopohjainen. Pahimmin tämä näkyy uusien olioiden luomisessa. Selvästikään asiakassovelluksessa ei ole mahdollista luoda olioita suoraan, koska järjestelmä ei osaisi päättää, mille mahdollisista palvelimista tämä olio luotaisiin eikä järjestelmä edes tiedä, miten olioiden luominen tapahtuisi. Tämä ongelma on helposti ohitettavissa luomalla yksi palvelun perusolio, jonka kautta muiden olioiden luominen tapahtuu. Omassa sovelluksessani tätä virkaa toimitti *CalendarService*-luokan olio.

Toinen huomio oli, että Corban kanssa työskennellessä täytyy käyttää määrittelyyn paljon enemmän aikaa, kuin normaalisti yhden ohjelmointikielen järjestelmää käytettäessä. Olin alun perin suunnitellut kalenteripalvelun toteutettavaksi Javalla, ja siksi en ollut suunnitellut esimerkiksi juuri teh-

dasluokkia. Huomasin ohjelmaa toteuttaessani myös muita puutteita joita suunnitelmaan oli jäänyt. Näiden korjaaminen myöhemmin oli työlästä, mutta erittäin hankalaksi tilanne muodostuisi, jos en olisi ainoa kehittäjä projektissa. Ja muutokset olisivat lähes mahdottomia, jos rajapinta ei olisi yhden organisaation vapaasti muuteltavissa. Eli Corban käyttäminen pakottaa tietynlaiseen huolellisuuteen ja pikkutarkkuuteen suunnitteluvaiheessa, mitä useinkin voidaan pitää lähes pelkästään hyvänä asiana.

5. Yhteenveto

Olen edellisissä kolmessa luvussa esitellyt syitä Corban käyttämisen sekä yleensä että erityisesti Javan kanssa. Lisäksi olen esittänyt eri lähteistä keräämiäni vertailuja, joiden avulla voidaan perustella Corban käyttämistä. Olen myös esitellyt Corban rakennetta ja siihen olennaisesti kuuluvia palvelumääritelmiä. Lopuksi kerroin omista kokemuksistani Corballa toteutetun kalenteripalvelun suunnittelusta ja toteuttamisesta.

Koska Corba on vain määritelmä, sen pohjalta voidaan toteuttaa useita kilpailevia tuotteita. Nämä tuotteet voidaan toteuttaa mihin tahansa laitteistoympäristöön ja mille tahansa käyttöjärjestelmälle, joten ympäristöriippumattoman ohjelmistoperheen tuottaminen on helpompaa. Kilpailu voi puolestaan vaikuttaa tuotteiden laatuun ja hintaan. Corba on myös hyvä ja teollisuuden laajalti hyväksymä määritelmä, joka byrokratian kankeudesta johtuen pysyy suhteellisen muuttumattomana.

Corba on myös täysin oliopohjainen järjestelmä, joten oliopohjaisten sovellusten toteuttaminen sen avulla on suoraviivaista ja kohtuullisen vaivatonta. Corban yhteyteen on määritelty lukuisa joukko yleisiä palveluita, joista jokaisesta on olemassa toteutus ja joista suuri osa voi auttaa ryhmäohjelmienkin toteuttamisessa.

Yleiskäyttöisenä järjestelmänä Corba tarjoaa kaikille järjestelmässä ajettaville olioille mahdollisuuden kommunikoida keskenään ja käyttää toistensa palveluita hyväkseen. Erityisen hyödylliseksi tämä voi muodostua Internetin tapaisessa maailmanlaajuisessa verkossa, jossa voisi kuvitella olevan mahdollista myydä Corban avulla käytettäväksi tarjottavia palveluita. Näin voitaisiin luopua järkälemäisistä kaikkea osaavista sovelluksista ja siirtyä erikoistuneisiin ja tehokkaisiin yhden asian osaajakomponentteihin kuten kalenteri- tai sähköpostiolioihin.

Omien kokemusteni perusteella voin sanoa, että Corban käyttäminen muuttaa ohjelmistosuunnittelun luonnetta hiukan. Rajapintojen suunnittelu nousee huomattavan tärkeäksi osaksi sovelluksen tuottamista ja tietoliiken-

teen ongelmat piiloutuvat kehittäjältä täysin. Corban luonne pakottaa käyttämään oliomalleja ja ottamaan huomioon poikkeukset muillakin kielillä kuin Javalla. Vanha sanonta: ”hyvin suunniteltu on puoliksi tehty”, pitää Corba-maailmassa todellakin paikkansa.

Kaikella edellä esitetyllä olen pyrkinyt etsimään vastausta kysymykseeni: kannattaako Corbaa käyttää ryhmäohjelmien toteuttamisvälineenä. Olen vahvasti sitä mieltä, että välittäjäohjelmistot tekevät sovelluksista paremmin toimivia, yleiskäyttöisiä ja helpommin toteutettavia. Ja lukemani ja kokemani perusteella Corba on erittäin kilpailukykyinen välittäjäohjelmistomääritelmä. Siksi olen valmis suosittelemaan Corban käyttöä sekä ryhmäohjelmien että kaikkien muidenkin kehittyneiden asiakas-palvelinsovellusten rakennusvälineenä.

Jatkotutkimuksissa voitaisiin tutkia, miten onnistuu Corba-palveluiden ja Internetiin liittyvien protokollien yhdistäminen laajemminkin; ryhmätyötä tukevista protokollista tällaisia voisivat olla sähköposti ja uutisryhmät, myös kalenteripalvelua voisi suunnitella tarkemmin ja ottaa huomioon sen erityisongelmat kuten aikavyöhykkeet ja eri aikoihin eri paikoissa alkavat kesä- ja talviajat. Myös Corban ja Java-appletien yhdistäminen World Wide Webissä älykkäiden agenttien avulla toimivaksi palveluolioiden verkoksi voisi olla hedelmällinen tutkimusalue.

Vain aika näyttää, saavatko IDL-pohjaiset palvelut niille kuuluvan arvostuksen vai vaipuuko Corba muiden hyvien mutta huonosti markkinoitujen ideoiden tavoin tietotekniikan unholaan.

Lähteet

[Chung *et al.*, 98] P.E.Chung, Y.Huang, S.Yajnik, D.Liang, J.Shih, C.-Y.Wang, Y.-M.Wang: DCOM and CORBA – Side by Side, Step by Step, and Layer by Layer. *C++ Report*, January 1998, vol.10, no.1; ss. 18–29,40

[CORBA, 96] *The Common Object Request Broker : Architecture and Specification. Revision 2.0.* OMG:n virallinen Corba-spesifikaatio, julkaistu ensimmäisen kerran heinäkuussa 1995, päivitetty heinäkuussa 1996. Saatavissa OMG:n kotisivulta ftp://ftp.omg.org/pub/docs/formal/97-02-25.pdf

[CORBA, 97] *Calendar Facility – Request for Proposal.* OMG:n vuoden 1997 lopulla julkistama lopullinen esityspyyntö kalenteripalvelun rajapintamääritelmäksi. Saatavissa eri muodoissa OMG:n kotisivulta; tässä käytetty PDF-versiota ftp://ftp.omg.org/pub/docs/bom/97-12-07.pdf

[CORBA, 98] *The Common Object Request Broker : Architecture and Specification. Revision 2.2.* Corba-spesifikaation päivitetty versio helmikuulta 1998.

[Curtis, 97] *Java, RMI and Corba; A White Paper.* Löydetty 2.6.1997 www:stä: http://www.omg.org/news/wpjava.htm

[Edwards, 97] Larry M. Edwards: Distributed Object Computing: The Evolution of Client-Server. *Databased Web Advisor*, May 1997, sivut 44–47.

[Javasoft, 99] *Sun Announces Availability of Java Hotspot Performance Engine.* Javasoftin lehdistötiedote 27. huhtikuuta 1999. Ladattu www:stä 7.5.1999: http://www.javasoft.com/pr/1999/04/pr990427-01.html

[Kähkipuro, 97] *CORBA 2.0-ohjelmointi.* Kurssimonisteet Pekka Kähkipuron pitämältä Tieturin kurssilta huhtikuussa 1997.

[Netscape, 97] Luku 8 Netscape Enterprise Serverin kehittäjän oppaasta. http://developer.netscape.com/docs/manuals/enterprise/javapg/caffn.htm

[OMG, 97] *Comparing ActiveX and CORBA/IIOP.* Artikkeli lähinnä ActiveX:n ongelmista. Ladattu WWW:stä 21.9.1998 osoitteesta http://www.omg.org/news/activex.htm

[OMG, 97/2] *CORBA Services : Common Object Services Specification*. Kuvaus Corba-standardiin kuuluvista ylimääräisistä palveluista. Päivitetty heinäkuussa 1997.

[Orfali *et al.*, 96] Robert Orfali, Dan Harkey and Jeri Edwards: *The Essential Distributed Objects Survival Guide*. Wiley, 1996

[Orfali *et al.*, 97] Robert Orfali, Dan Harkey and Jeri Edwards: *Instant CORBA*. Wiley, 1997.

[Orfali & Harkey, 98] Robert Orfali and Dan Harkey: *Client/Server Programming with JAVA and Corba, Second Edition*. Wiley, 1998.

[Otte *et al.*, 96] Randy Otte, Paul Patrick and Mark Roy: *Understanding CORBA – The Common Object Request Broker Architecture*. Prentice Hall, 1996.

Liite 1: Sanasto

Corban terminologia on useimmille asiaan perehtymättömille täysin uutta. Tähän on kerätty termejä, joihin Corba-julkaisuja lukiessa törmää. Suomenkieliset termit on suurimmaksi osaksi otettu Pekka Kähkipuron kurssimonisteesta [Kähkipuro/97].

BOA	Basic Object Adapter eli Oliosovitin. Sijaitsee ORBin ytimen päällä ja ottaa vastaan palvelupyyntöjä palvelimella olevien olioiden puolesta. Luo olioille olioviitteet eli ID:t. Oliosovitin myös rekisteröi luokkansa ja niiden instanssit *toteutustietokantaan*.
Corba	Common Object Request Broker Architecture eli Yhteinen oliopyyntöjen välittäjä –arkkitehtuuri.
Implementation Repository	Toteutustietokanta. Ylläpitää ajonaikaista varastoa palvelimen tukemien luokkien tiedoista, instantioiduista olioista sekä näiden ID:istä.
Järjestely	Katso marshaling, unmarshaling
Marshaling, unmarshaling	Tiedon pakkaaminen siirrettävään muotoon lähettävässä sovelluksessa tai sen purkaminen vastaanottavassa päässä. Tietoa siirrettäessä pakataan palvelinolion tyyppi, metodien parametrit, paluuarvot ja poikkeukset. Tekstissä käytettävä termi "järjestely" on oma suomennokseni, koska tälle termille ei ole vakiintunutta virallista suomennosta.
Oliopyyntöjen välittäjä	Katso ORB.

Oliosovitin	Katso BOA.
ORB	Object Request Broker eli oliopyyntöjen välittäjä. Corba-toteutuksen ydin. Vastaanottaa asiakassovelluksilta tulevia metodikutsuja ja välittää ne kohdeoliolle.
POA	Portable Object Adapter. BOA:n paranneltu versio, jonka tärkeimpänä ominaisuutena on oliototeutusten toimiminen myös eri ORBissa kuin mihin se on alun perin kehitetty.
Runko	Katso Skeleton
Skeleton	Myös IDL Skeleton tai IDL runko. Olion ilmentymä palvelimen puolella. Järjestelee metodikutsussa parametrit palvelinolion ymmärtämään muotoon ja kutsuu varsinaista toteutusta. Metodin suorituksen jälkeen järjestelee paluuarvot siirrettävään muotoon.
Stub	Myös IDL Stub tai IDL tynkä. Olion ilmentymä asiakassovelluksessa. Järjestelee metodia kutsuttaessa parametrit ja paluuarvot.
Toteutustietokanta	Katso Implementation Repository.
Tynkä	Katso Stub